

# Introducción a la programación.

## Caso práctico

La evolución de Internet y de las nuevas tecnologías, así como las diferentes posibilidades para establecer nuevas líneas de negocio para la empresa BK Programación, han hecho que **Ada** haya decidido abrir una vía de innovación. Para ello, su empresa deberá realizar el desarrollo de sus aplicaciones a través de lenguajes y técnicas de programación modernos, aunque con una eficiencia y flexibilidad contrastadas.

**María y Juan**, ayudados y orientados por **Ada**, recordarán y ampliarán sus conocimientos relacionados con la programación, permitiéndoles crear software que pueda adaptarse a nuevas situaciones, como el funcionamiento en diferentes plataformas (PDA, Móviles, Web, etc.) o la interacción con bases de datos. Todo ello sin perder de vista de dónde parten y hacia dónde quieren redirigir sus esfuerzos.

Estas innovaciones, junto a la predisposición para adaptarse y evolucionar que BK Programación está potenciando en todas sus áreas, repercutirán en una mayor capacidad de respuesta ante las necesidades de sus posibles clientes. En definitiva, conseguir mayor competitividad.



ITE (CC BY-NC-SA)



[Ministerio de Educación y Formación Profesional. \(Dominio público\)](#)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Introducción.

---

¿Cuántas acciones de las que has realizado hoy, crees que están relacionadas con la programación? Hagamos un repaso de los primeros instantes del día: te ha despertado la alarma de tu teléfono móvil o radio-despertador, has preparado el desayuno utilizando el microondas, mientras desayunabas has visto u oído las últimas noticias a través de tu receptor de televisión digital terrestre, te has vestido y puede que hayas utilizado el ascensor para bajar al portal y salir a la calle, etc. Quizá no es necesario que continuemos más para darnos cuenta de que casi todo lo que nos rodea, en alguna medida, está relacionado con la programación, los programas y el tratamiento de algún tipo de información.

El volumen de datos que actualmente manejamos y sus innumerables posibilidades de tratamiento constituyen un vasto territorio en el que los programadores tienen mucho que decir.



Rüdiger Wölk (CC BY-SA)

En esta primera unidad realizaremos un recorrido por los conceptos fundamentales de la programación de aplicaciones. Iniciaremos nuestro camino conociendo con qué vamos a trabajar, qué técnicas podemos emplear y qué es lo que pretendemos conseguir. Continuaremos con el análisis de las diferentes formas de programación existentes, identificaremos qué fases conforman el desarrollo de una aplicación software y avanzaremos detallando las características relevantes de cada uno de los lenguajes de programación disponibles, para posteriormente, realizar una visión general del lenguaje de programación Java. Finalmente, tendremos la oportunidad de conocer con qué herramientas podríamos desarrollar nuestros programas, escogiendo entre una de ellas para ponernos manos a la obra utilizando el lenguaje Java.

## 2.- Programas y programación.

### Caso práctico

**Ada** conoce bien lo que significa tener que llevar a cabo el proceso completo de creación de software y sabe que, en ocasiones, no se le da la importancia que debería a las fases iniciales de este proceso. Quiere que **Juan**, que desarrolla programas casi sin darse cuenta, recuerde las ventajas que aporta un buen análisis inicial de los problemas a solucionar y que no aborde el desarrollo de sus programas sentándose directamente ante el ordenador a teclear código.



Ministerio de Educación y FP ([CC BY-NC](#))

**Juan** le comenta a **Ada** y a **María**: —La verdad es que cuando conoces bien un lenguaje de programación crees que puedes hacer cualquier programa directamente sobre el ordenador, pero al final te das cuenta de que deberías haberte parado a planificar tu trabajo. Muchas veces tienes que volver atrás, recodificar y en ocasiones, rehacer gran parte del programa porque lo que tienes no está bien planteado. Ocurre algo parecido en el desarrollo de otros productos o servicios: ¿os imagináis que la construcción de una casa no pase por la planificación de un arquitecto sino que sean los propios obreros los que tomen decisiones sobre la marcha?.

**María**, que permanece atenta a lo que dicen **Ada** y **Juan**, quiere aprender bien desde el principio y tendrá la ventaja de tener a su lado a dos expertos.

## 2.1.- Buscando una solución.

Generalmente, la primera razón que mueve a una persona hacia el aprendizaje de la programación es utilizar el ordenador como herramienta para resolver problemas concretos. Como en la vida real, la búsqueda y obtención de una solución a un problema determinado, utilizando medios informáticos, se lleva a cabo siguiendo unos pasos fundamentales. En la siguiente tabla podemos ver estas analogías.



Filosofías Filosóficas (CC BY-SA)

### Resolución de problemas

En la vida real...	En Programación...
Observación de la situación o problema.	<b>Análisis del problema:</b> requiere que el problema sea definido y comprendido claramente para que pueda ser analizado con todo detalle.
Pensamos en una o varias posibles soluciones.	<b>Diseño o desarrollo de algoritmos:</b> se establece una solución al problema sin entrar en detalles tecnológicos. Se aplican diferentes técnicas y principios para establecer de forma detallada los pasos a seguir para resolver el problema.
Aplicamos la solución que estimamos más adecuada.	<b>Resolución del algoritmo elegido en la computadora:</b> consiste en convertir el algoritmo en programa, ejecutarlo y comprobar que soluciona verdaderamente el problema.

¿Qué virtudes debería tener nuestra solución?

- ✓ **Corrección y eficacia:** si resuelve el problema adecuadamente.
- ✓ **Eficiencia:** si lo hace en un tiempo mínimo y con un uso óptimo de los recursos del sistema.

Para conseguirlo, cuando afrontemos la construcción de la solución tendremos que tener en cuenta los siguientes conceptos:

1. **Abstracción:** se trata de realizar un análisis del problema para descomponerlo en problemas más pequeños y de menor complejidad, describiendo cada uno de ellos de manera precisa. **Divide y vencerás**, esta suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.
2. **Encapsulación:** consiste en ocultar la información que manejan los diferentes elementos que forman el sistema. La forma de manejar esa información no debe influir en el resto de elementos del sistema.
3. **Modularidad:** un proyecto software será dividido en módulos independientes, dependiendo de su tamaño, donde cada uno de ellos tendrá su función correspondiente. Los demás módulos del sistema podrán utilizar su funcionalidad sin necesidad de conocer cómo funciona internamente.

### Citas para pensar

*Roger Pressman:* “El comienzo de la sabiduría para un ingeniero de software es reconocer la diferencia entre hacer que un programa funcione y conseguir que lo haga correctamente.”

## 2.2.- Algoritmos y programas.

Después de analizar en detalle el problema a solucionar, hemos de diseñar y desarrollar el algoritmo adecuado. Pero, ¿Qué es un algoritmo?

**Algoritmo:** secuencia ordenada de pasos, descrita sin ambigüedades, que conducen a la solución de un problema dado.

Los algoritmos son independientes de los lenguajes de programación y de las computadoras donde se ejecutan. Un mismo algoritmo puede ser expresado en diferentes lenguajes de programación y podría ser ejecutado en diferentes dispositivos. Piensa en una receta de cocina, ésta puede ser expresada en castellano, inglés o francés, podría ser cocinada en fogón o vitrocerámica, por un cocinero o más, etc. Pero independientemente de todas estas circunstancias, el plato se preparará siguiendo los mismos pasos.

La diferencia fundamental entre algoritmo y **programa** es que, en el segundo, los pasos que permiten resolver el problema, deben escribirse en un determinado **lenguaje de programación** para que puedan ser ejecutados en el ordenador y así obtener la solución.



Stockbyte DVD-CD (CC BY-NC)

**Los lenguajes de programación** son sólo un medio para expresar el algoritmo, es decir, establece una serie de normas sintácticas y semánticas para expresarlo. El diseño de los algoritmos será una tarea que necesitará de la creatividad del desarrollador y de los conocimientos de las técnicas de programación. Estilos distintos, de distintos programadores a la hora de obtener la solución del problema, darán lugar a algoritmos diferentes, igualmente válidos.

En esencia, todo problema se puede describir por medio de un algoritmo y las características fundamentales que éstos deben cumplir son:

- ✓ Debe ser **preciso** e indicar el orden de realización paso a paso.
- ✓ Debe estar **definido**, si se ejecuta dos o más veces con los mismos datos de entrada, debe obtener el mismo resultado cada vez. Además, debe dar una respuesta a cualquier dato de entrada.
- ✓ Debe ser **finito**, debe tener un número finito de pasos.

Para representar gráficamente los algoritmos que vamos a diseñar, tenemos a nuestra disposición diferentes herramientas que ayudarán a describir su comportamiento de una forma precisa y genérica, para luego poder codificarlos con el lenguaje que nos interese. Entre otras tenemos:

- ✓ **Diagramas de flujo:** Esta técnica utiliza símbolos gráficos para la representación del algoritmo. Suele utilizarse en las fases de análisis.
- ✓ **Pseudocódigo:** Esta técnica se basa en el uso de palabras clave en lenguaje natural, ..... constantes, ..... variables, otros objetos, instrucciones y estructuras de programación que expresan de forma escrita la solución del problema. Es la técnica más utilizada actualmente.
- ✓ **Tablas de decisión:** En una tabla son representadas las posibles condiciones del problema con sus respectivas acciones. Suele ser una técnica de apoyo al pseudocódigo cuando existen situaciones condicionales complejas.

### Debes conocer

A continuación te ofrecemos algunos recursos interesantes:

- ✓ [Elementos visuales de los diagramas de flujo](#): En este enlace podrás aprender los elementos gráficos más utilizados en la construcción de diagramas de flujo.
- ✓ **Software DFD:** Se trata de una aplicación que permite construir diagramas de flujo de forma gráfica. Es una aplicación portable: tras su descarga tan solo tienes que ejecutar la aplicación.
  - ◆ Descárgala [aquí](#).
  - ◆ Observa en el siguiente vídeo su funcionamiento.

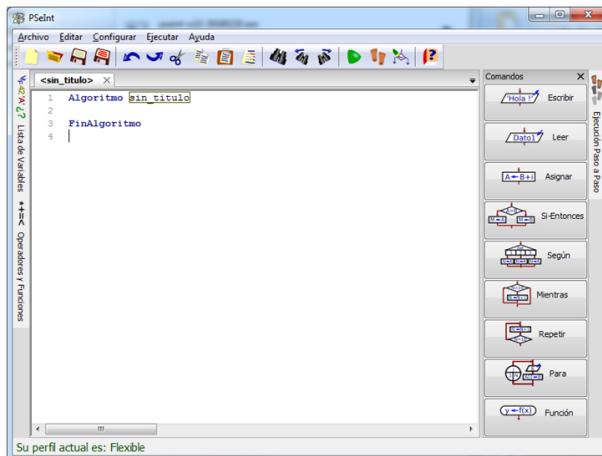
<https://www.youtube.com/embed/tScN7c27olM>

[Descripción Textual Alternativa para el video "DFD"](#)

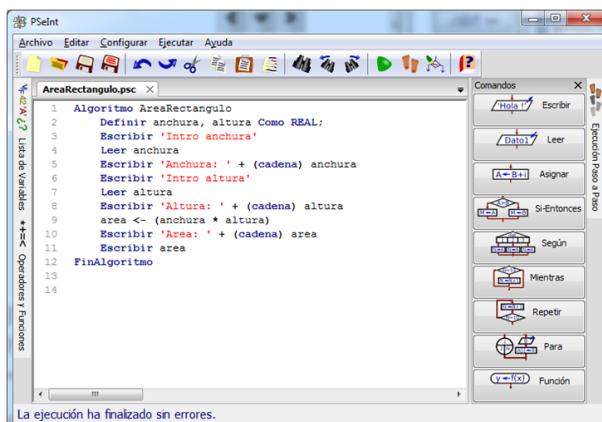
- ✓ **Pseint:** Se trata de una aplicación que permite la construcción de algoritmos a través de un fácil e intuitivo pseudocódigo, complementado con un editor de diagramas de flujo. Es una de las herramientas más utilizadas para iniciarse en el mundo de la algoritmia, proporcionando un entorno con numerosas

ayudas y recursos didácticos.

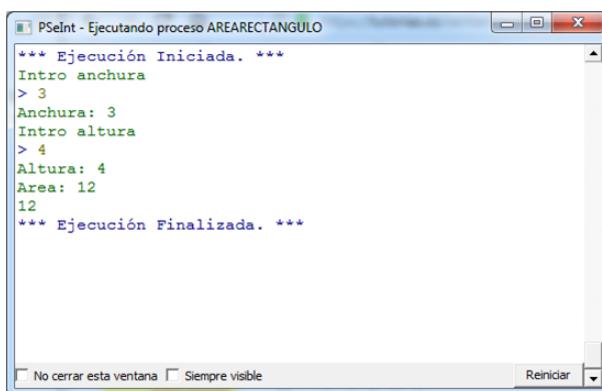
Como vemos en la siguiente, disponemos de 2 paneles principales, el central para incorporar pseudocódigo directamente, y el de la derecha, con los comandos disponibles para ayudar, si se desconoce la sintaxis en la escritura de instrucciones o estructuras de control.



Ministerio de Educación y FP (CC BY-NC)



Ministerio de Educación y FP (CC BY-NC)



Ministerio de Educación y FP (CC BY-NC)

En la segunda imagen se puede ver un ejemplo de pseudocódigo en Pseint que calcula el área de un cuadrado. Observa como las instrucciones están escritas en un pseudolenguaje bastante parecido al lenguaje natural. Los elementos del panel derecho permiten la introducción de código de forma alternativa a la escritura de sentencias en pseudocódigo.

Por último, la tercera imagen muestra el resultado de interpretar el algoritmo tras pulsar el botón verde de reproducción de la barra de tareas.

Toda la información sobre Pseint.

[Pseint](#)

## Autoevaluación

Rellena los huecos con los conceptos adecuados:

A los pasos que permiten resolver el problema, escritos en un lenguaje de programación, para que puedan ser ejecutados en el ordenador y así obtener la solución, se les denomina:  .

Enviar

A los pasos que permiten resolver el problema, escritos en un lenguaje de programación, para que puedan ser ejecutados en el ordenador y así obtener la solución, se les denomina: **Programa**. Si estos pasos estuvieran descritos en un lenguaje genérico independiente de la máquina y del lenguaje de programación, estaríamos hablando de **algoritmos**.

### 3.- Paradigmas de la programación.

#### Caso práctico

**Ada** comenta con **Juan** y **María** los distintos enfoques para el desarrollo de programas que han existido a lo largo de la historia de la programación, destacando que todos van a tener que “renovar” su forma de pensar, si quieren comenzar a utilizar un lenguaje moderno que les permita construir programas adaptados a las nuevas necesidades de sus clientes.



Stockbyte CD-DVD Num. CD165 [CC BY-NC](#)



[barraquito from Santa Cruz de Tenerife, Canary Islands, Spain \(CC BY-SA\)](#)

¿Cuántas formas existen de hacer las cosas? Supongo que estarás pensando: varias o incluso, muchas. Pero cuando se establece un patrón para la creación de aplicaciones nos estamos acercando al significado de la palabra paradigma. Si establecemos una serie de normas y principios que recojan experiencia y buenas prácticas de otros desarrolladores para su uso en la resolución de problemas, estaremos creando un paradigma de programación.

**Paradigma de programación:** es un modelo básico para el diseño y la implementación de programas. Este modelo determinará como será el proceso de diseño y la estructura final del programa.

El paradigma representa un enfoque particular o filosofía para la construcción de software. Cada uno tendrá sus ventajas e inconvenientes, será más o menos apropiado, pero no es correcto decir que exista uno mejor que los demás. Algunos de ellos son:

- **Programación Declarativa:** Se basa en el desarrollo de algoritmos aplicando una especificación de un conjunto de condiciones, proposiciones, afirmaciones y restricciones que describen el problema. Las sentencias utilizadas describen el problema que se quiere solucionar, pero no la instrucciones necesarias para llegar a la solución. El lenguaje SQL está basado en este paradigma. Dentro de este paradigma se encuentran la programación funcional y la programación lógica.
- **Programación Imperativa:** Se basa en el desarrollo de algoritmos detallando de forma clara y específica los comandos a ejecutar para, a través del paso por diferentes estados, llegar a la solución. Se basa en el uso de variables, tipos de datos, expresiones y estructuras de control del flujo de ejecución. Lenguajes como Python, Java, C++, C# ... son lenguajes imperativos. Dentro de este paradigma se encuentran la programación convencional (programación no estructurada), la programación estructurada, la programación orientada a objetos, la programación orientada a eventos, la programación orientada a aspectos ...

Existen múltiples paradigmas, incluso puede haber lenguajes de programación que no se clasifiquen únicamente dentro de uno de ellos. Un lenguaje como Smalltalk es un lenguaje basado en el paradigma orientado a objetos. El lenguaje de programación Scheme, en cambio, soporta sólo programación funcional. Python, soporta múltiples paradigmas.

## Para saber más

Te proponemos el siguiente enlace en el que encontrarás información adicional sobre los diferentes paradigmas de programación.

[Paradigmas de programación y lenguajes](#)

¿Cuál es el objetivo que se busca con la aplicación de los diferentes enfoques? Fundamentalmente, reducir la dificultad para el desarrollo y mantenimiento de las aplicaciones, mejorar el rendimiento del programador, reutilizando código en la medida de lo posible y, en general, mejorar la productividad y calidad de los programas.

## Autoevaluación

¿En qué paradigma de programación podríamos enmarcar el lenguaje de programación Java?

- Programación Estructurada.
- Programación Declarativa.
- Programación Orientada a Objetos.

No, Java permite trabajar con una filosofía más potente que la programación estructurada.

No, la programación declarativa se encarga de describir el problema y no las sentencias para su solución.

Sí, Java emplea la filosofía de ver el mundo como objetos que tienen propiedades y métodos que les permiten interactuar entre ellos.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

## 4.- Fases de la programación.

### Caso práctico

**Juan** pregunta a **Ada** cómo van a realizar todo el proceso de producción, y duda si el utilizar un nuevo lenguaje supondrá cambiar drásticamente los métodos aprendidos en el pasado.

**Ada** tranquiliza a **Juan** y a **María**: —Está claro que las fases principales que hemos estado llevando a cabo a lo largo de nuestros anteriores proyectos se seguirán aplicando, aunque con algunas diferencias. Lo más importante Juan, es que sigamos adecuadamente el método de trabajo para conseguir buenos resultados.— ¿Me costará mucho trabajo adaptarme? —pregunta **María**.

**Ada** le contesta sentándose a su lado: —No te preocupes **María**, se trata de adaptar conocimientos que ya tienes y aprender algunos otros.



Ministerio de Educación y FP (CC BY-NC)

Sea cual sea el estilo que escojamos a la hora de automatizar una determinada tarea, debemos realizar el proceso aplicando un método a nuestro trabajo. Es decir, sabemos que vamos a dar solución a un problema, aplicando una filosofía de desarrollo y lo haremos dando una serie de pasos que deben estar bien definidos.

El proceso de creación de software puede dividirse en diferentes fases:

- ✓ **Fase de resolución del problema.**
- ✓ **Fase de implementación.**
- ✓ **Fase de explotación y mantenimiento.**

A continuación, analizaremos cada una de ellas.

## 4.1.- Resolución del problema.

Para el comienzo de esta fase, es necesario que el problema sea definido y comprendido claramente para que pueda ser analizado con todo detalle. A su vez, la fase de resolución del problema puede dividirse en dos etapas:

### a. Análisis

Por lo general, el análisis indicará la especificación de requisitos que se deben cubrir. Los contactos entre el analista/programador y el cliente/usuario serán numerosos, de esta forma podrán ser conocidas todas las necesidades que precisa la aplicación. Se especificarán los procesos y estructuras de datos que se van a emplear. La creación de prototipos será muy útil para saber con mayor exactitud los puntos a tratar.

El análisis inicial ofrecerá una idea general de lo que se solicita, realizando posteriormente sucesivos refinamientos que servirán para dar respuesta a las siguientes cuestiones:

- ✓ ¿Cuál es la información que ofrecerá la resolución del problema?.
- ✓ ¿Qué datos son necesarios para resolver el problema?.

La respuesta a la primera pregunta se identifica con los resultados deseados o las salidas del problema. La respuesta a la segunda pregunta indicará qué datos se proporcionan o las entradas del problema.

En esta fase debemos aprender a analizar la documentación de la empresa , investigar, observar todo lo que rodea el problema y recopilar cualquier información útil.



Ildar Sagdejev (CC BY-SA)

### Ejercicio resuelto

Vamos a ilustrar esta fase realizando el análisis del siguiente problema:

“Leer el radio de un círculo y calcular e imprimir su superficie y circunferencia.”

Está claro que las entradas de datos en este problema se reducen al radio del círculo, pero piensa ¿qué salidas de datos ofrecerá la solución?

[Mostrar retroalimentación](#)

Las salidas serán...

Variable de salida SUPERFICIE: será la superficie del círculo. (¿Te acuerdas? El número Pi por el radio al cuadrado).

Variable de salida CIRCUNFERENCIA: será la longitud de la circunferencia del círculo. (¿Y de ésta? Dos por el número Pi y por el radio)

Y la entrada...

Variable RADIO: será el radio del círculo.

Estas variables RADIO, SUPERFICIE y CIRCUNFERENCIA podrán ser de tipo real (números con parte entera y parte decimal, por ejemplo: 3,57)

### b. Diseño

En esta etapa se convierte la especificación realizada en la fase de análisis en un diseño más detallado, indicando el comportamiento o la secuencia lógica de instrucciones capaz de resolver el problema planteado. Estos pasos sucesivos, que indican las instrucciones a ejecutar por la máquina, constituyen lo que conocemos como algoritmo.

Durante la fase de diseño, se plantea la aplicación a desarrollar como una única operación global, y se va descomponiendo en operaciones más sencillas, detalladas y específicas. El nivel de descomposición dependerá del tamaño del problema. En cada nivel de refinamiento, las operaciones identificadas se asignan a módulos separados.

Hay que tener en cuenta que antes de pasar a la implementación del algoritmo, hemos de asegurarnos que tenemos una solución adecuada. Para ello, todo diseño requerirá de la realización de la **prueba o traza** del programa. Este

proceso consistirá en un seguimiento paso a paso de las instrucciones del algoritmo utilizando datos concretos. Si la solución aportada tiene errores, tendremos que volver a la fase de análisis para realizar las modificaciones necesarias o tomar un nuevo camino para la solución. Sólo cuando el algoritmo cumpla los requisitos y objetivos especificados en la fase de análisis se pasará a la fase de implementación.

## 4.2.- Implementación.

Si la fase de resolución del problema requiere un especial cuidado en la realización del análisis y el posterior diseño de la solución, la fase de implementación cobra también una especial relevancia. Llevar a la realidad nuestro algoritmo implicará cubrir algunas etapas más que se detallan a continuación.

### a. Codificación o construcción

Esta etapa consiste en transformar o traducir los resultados obtenidos a un determinado lenguaje de programación. Para comprobar la calidad y estabilidad de la aplicación se han de realizar una serie de pruebas que comprueben las funciones de cada módulo (pruebas unitarias), que los módulos funcionan bien entre ellos (pruebas de interconexión) y que todos funcionan en conjunto correctamente (pruebas de integración).



Stockbyte CD-DVD Num. V07 [\(CC BY-NC\)](#)

Cuando realizamos la traducción del algoritmo al lenguaje de programación debemos tener en cuenta las reglas gramaticales y la sintaxis de dicho lenguaje. Obtendremos entonces el código fuente, lo que normalmente conocemos por programa.

Pero para que nuestro programa comience a funcionar, antes debe ser traducido a un lenguaje que la máquina entienda. Este proceso de traducción puede hacerse de dos formas, compilando o interpretando el código del programa.

**Compilación:** Es el proceso por el cual se traducen las instrucciones escritas en un determinado lenguaje de programación a lenguaje que la máquina es capaz de interpretar, normalmente código binario.

El proceso de compilación se puede llevar a cabo de dos formas:

- ✓ **A través de un compilador:** programa informático que realiza la traducción. Recibe el código fuente, realiza un análisis lexicográfico, semántico y sintáctico, genera un código intermedio no optimizado, optimiza dicho código y finalmente, genera el código objeto/máquina ejecutable en una plataforma específica.
- ✓ **A través de un Intérprete:** programa informático capaz de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel. Los intérpretes se diferencian de los compiladores en que mientras éstos traducen un programa escrito en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

Una vez traducido, sea a través de un proceso de compilación o de interpretación, el programa podrá ser ejecutado.

### b. Prueba de ejecución y validación

Para esta etapa es necesario implantar la aplicación en el sistema donde va a funcionar, debe ponerse en marcha y comprobar si su funcionamiento es correcto. Utilizando diferentes datos de prueba se verá si el programa responde a los requerimientos especificados, si se detectan nuevos errores, si éstos son bien gestionados y si la interfaz es amigable. Se trata de poner a prueba nuestro programa para ver su respuesta en situaciones difíciles.

Mientras se detecten errores y éstos no se subsanen no podremos avanzar a la siguiente fase. Una vez corregido el programa y testeado se documentará mediante:

- ✓ **Documentación interna:** Encabezados, descripciones, declaraciones del problema y comentarios que se incluyen dentro del código fuente.
- ✓ **Documentación externa:** Son los manuales que se crean para una mejor ejecución y utilización del programa.

## Autoevaluación

### Rellena los huecos con los conceptos adecuados:

En la fase de codificación, hemos de tener en cuenta la  del lenguaje para obtener el código fuente o programa. Posteriormente, éste deberá ser  o  para que pueda ser ejecutado posteriormente.

La **sintaxis** y reglas gramaticales del lenguaje de programación que estemos utilizando deben ser respetadas para obtener un código fuente correcto. Este código fuente debe ser **compilado o interpretado**.

**interpretado**, utilizando un programa compilador o intérprete, para transformarlo a un formato que sea ejecutable por la máquina.

## 4.3.- Explotación.

---

Cuando el programa ya está instalado en el sistema y está siendo de utilidad para los usuarios, decimos que se encuentra en fase de explotación.

Periódicamente será necesario realizar evaluaciones y, si es necesario, llevar a cabo modificaciones para que el programa se adapte o actualice a nuevas necesidades, pudiendo también corregirse errores no detectados anteriormente. Este proceso recibe el nombre de mantenimiento del software.

Será imprescindible añadir una documentación adecuada que facilite al programador la comprensión, uso y modificación de dichos programas.



Stockbyte CD-DVD Num. CD109 ([CC BY-NC](#))

## 5.- Ciclo de vida del software.

### Caso práctico

**María** le pregunta a **Juan**: —¿Juan, qué ocurre cuando terminas un programa? ¿Se entrega al cliente y ya está? La verdad es que los programas que he hecho han sido para uso propio y no sé cómo termina el proceso con los clientes.

Contesta **Juan**: —Pues verás, cuando terminas un programa, o crees que lo has terminado, hay que llevar a cabo toda clase de pruebas para ver dónde puede fallar. Después mejoras los posibles fallos y posteriormente se entrega al cliente, ahí es donde ves si tu software ha sido bien construido. El cliente lo utilizará y durante un tiempo puede ser que haya que arreglar alguna cosilla. Y cuando ya está todo correcto, en ocasiones, se establece un contrato de mantenimiento con el cliente. Como ves, desarrollar software no consiste sólo en programar y ya está.



Ministerio de Educación y FP [\(CC BY-NC\)](#)

Sean cuales sean las fases en las que realicemos el proceso de desarrollo de software, y casi independientemente de él, siempre se debe aplicar un modelo de ciclo de vida.

**Ciclo de vida del software:** es una sucesión de estados o fases por las cuales pasa un software a lo largo de su "vida".

El proceso de desarrollo puede involucrar siempre las siguientes etapas mínimas:

- ✓ Especificación y Análisis de requisitos.
- ✓ Diseño.
- ✓ Codificación.
- ✓ Pruebas.
- ✓ Instalación y paso a Producción.
- ✓ Mantenimiento.

Aprenderás mucho más sobre el ciclo de vida del software en el Módulo Profesional "Entornos de Desarrollo".

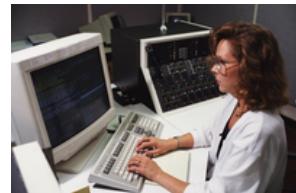
## 6.- Lenguajes de programación.

### Caso práctico

**Ada y Juan** están recordando lo complejos que eran algunos lenguajes de programación, **Ada** comenta: —Cuando yo empecé en esto, había relativamente pocos lenguajes de programación y no permitían hacer programas como los que ahora desarrollamos.

**Juan** indica que él conoce las características generales de algunos lenguajes, pero que le gustaría saber algo más sobre los que hubo, hay y habrá.

**Maria** que asiente con la cabeza, piensa que aprender más sobre los lenguajes disponibles en la actualidad puede ayudar a la hora de elegir entre unos u otros.



Stockbyte (DVD-CD) Num. V07 ([CC BY-NC](#))

Como hemos visto, en todo el proceso de resolución de un problema mediante la creación de software, después del análisis del problema y del diseño del algoritmo que pueda resolverlo, es necesario traducir éste a un lenguaje que exprese claramente cada uno de los pasos a seguir para su correcta ejecución. Este lenguaje recibe el nombre de lenguaje de programación.

**Lenguaje de programación:** Conjunto de reglas sintácticas y semánticas, símbolos y palabras especiales establecidas para la construcción de programas. Es un lenguaje artificial, una construcción mental del ser humano para expresar programas.

Además, cada lenguaje de programación, al igual que otro tipo de lenguajes, se basa en una gramática.

**Gramática del lenguaje:** Reglas aplicables al conjunto de símbolos y palabras especiales del lenguaje de programación para la construcción de sentencias correctas.

Además, cada gramática dispone de:

1. **Léxico:** Es el conjunto finito de símbolos y palabras especiales, es el vocabulario del lenguaje.
2. **Sintaxis:** Son las posibles combinaciones de los símbolos y palabras especiales. Está relacionada con la forma de los programas.
3. **Semántica:** Es el significado de cada construcción del lenguaje, la acción que se llevará a cabo.

Hay que tener en cuenta que pueden existir sentencias sintácticamente correctas, pero semánticamente incorrectas. Por ejemplo, “*Un aveSTRUZ dio un zarpazo a su cuidador*” está bien construida sintácticamente, pero es evidente que semánticamente no.

Una característica relevante de los lenguajes de programación es, precisamente, que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos. A través de este conjunto se puede lograr la construcción de un programa de forma colaborativa.

Los lenguajes de programación pueden ser clasificados en función de lo cerca que estén del lenguaje humano o del lenguaje de los computadores. El lenguaje de los computadores son los códigos binarios, es decir, secuencias de unos y ceros. Detallaremos seguidamente las características principales de los lenguajes de programación.



Autor desconocido. (Dominio público)

## 6.1.- Lenguaje máquina.

Este es el lenguaje utilizado directamente por el procesador, consta de un conjunto de instrucciones codificadas en binario. Es el sistema de códigos directamente interpretable por un círculo microprogramable.

Este fue el primer lenguaje utilizado para la programación de computadores. De hecho, cada máquina tenía su propio conjunto de instrucciones codificadas en ceros y unos. Cuando un algoritmo está escrito en este tipo de lenguaje, decimos que está en código máquina.

Programar en este tipo de lenguaje presentaba los siguientes inconvenientes:

- ✓ Cada programa era válido sólo para un tipo de procesador u ordenador.
- ✓ La lectura o interpretación de los programas era extremadamente difícil y, por tanto, insertar modificaciones resultaba muy costoso.
- ✓ Los programadores de la época debían memorizar largas combinaciones de ceros y unos, que equivalían a las instrucciones disponibles para los diferentes tipos de procesadores.
- ✓ Los programadores se encargaban de introducir los códigos binarios en el computador, lo que provocaba largos tiempos de preparación y posibles errores.

A continuación, se muestran algunos códigos binarios equivalentes a las operaciones de suma, resta y movimiento de datos en lenguaje máquina.

### Algunas operaciones en lenguaje máquina.

Operación	Lenguaje máquina
SUMAR	00101101
RESTAR	00010011
MOVER	00111010

Dada la complejidad y dificultades que ofrecía este lenguaje, fue sustituido por otros más sencillos y fáciles utilizar. No obstante, hay que tener en cuenta que todos los programas para poder ser ejecutados, han de traducirse siempre al lenguaje máquina que es el único que entiende la computadora.

### Para saber más

Como recordatorio, te proponemos el siguiente enlace sobre cómo funciona el sistema binario.

[https://www.youtube.com/embed/cJhy9JutK\\_4](https://www.youtube.com/embed/cJhy9JutK_4)

[Resumen textual alternativo](#)

### Autoevaluación

**Rellena los huecos con los conceptos adecuados:**

En el lenguaje máquina de algunos procesadores, la combinación 00101101 equivale a la operación de

La Suma, resta y la operación de movimiento de datos eran muy utilizadas en los programas escritos en lenguaje máquina. Aún no se había extendido el uso de estructuras de programación como las sentencias condicionales o los bucles.

## 6.2.- Lenguaje Ensamblador.

La evolución del lenguaje máquina fue el lenguaje ensamblador. Las instrucciones ya no son secuencias binarias, se sustituyen por códigos de operación que describen una operación elemental del procesador. Es un lenguaje de bajo nivel, al igual que el lenguaje máquina, ya que dependen directamente del hardware donde son ejecutados. Normalmente una instrucción en ensamblador se corresponde con una instrucción de lenguaje máquina.

**Mnemotécnico:** son palabras especiales, que sustituyen largas secuencias de ceros y unos, utilizadas para referirse a diferentes operaciones disponibles en el juego de instrucciones que soporta cada máquina en particular.

En ensamblador, cada instrucción (mnemotécnico) se corresponde a una instrucción del procesador. En la siguiente tabla se muestran algunos ejemplos.

**Algunas operaciones y su mnemotécnico en lenguaje Ensamblador.**

Operación	Lenguaje Ensamblador
<b>MULTIPLICAR</b>	MUL
<b>DIVIDIR</b>	DIV
<b>MOVER</b>	MOV

En el siguiente gráfico puedes ver parte de un programa escrito en lenguaje ensamblador. En color rojo se ha resaltado el código máquina en ..... hexadecimal, en magenta el código escrito en ensamblador y en azul, las direcciones de memoria donde se encuentra el código.



Pero aunque ensamblador fue un intento por aproximar el lenguaje de los procesadores al lenguaje humano, presentaba múltiples dificultades:

- ✓ Los programas seguían dependiendo directamente del hardware que los soportaba.
- ✓ Los programadores tenían que conocer detalladamente la máquina sobre la que programaban, ya que debían hacer un uso adecuado de los recursos de dichos sistemas.
- ✓ La lectura, interpretación o modificación de los programas seguía presentando dificultades.

Todo programa escrito en lenguaje ensamblador necesita de un intermediario, que realice la traducción de cada una de las instrucciones que componen su código al lenguaje máquina correspondiente. Este intermediario es el programa ensamblador. El programa original escrito en lenguaje ensamblador constituye el código fuente y el programa traducido al lenguaje máquina se conoce como programa objeto que será directamente ejecutado por la computadora.

## 6.3.- Lenguajes compilados.

Para paliar los problemas derivados del uso del lenguaje ensamblador y con el objetivo de acercar la programación hacia el uso de un lenguaje más cercano al humano que al del computador, nacieron los lenguajes compilados. Algunos ejemplos de este tipo de lenguajes son: Pascal, Fortran, Algol, C, C++, etc.

Al ser lenguajes más cercanos al humano, también se les denomina **lenguajes de alto nivel**. Son más fáciles de utilizar y comprender, las instrucciones que forman parte de estos lenguajes utilizan palabras y signos reconocibles por el programador.

¿Cuáles son sus **ventajas**?



Leonardo da Vinci (Dominio público)

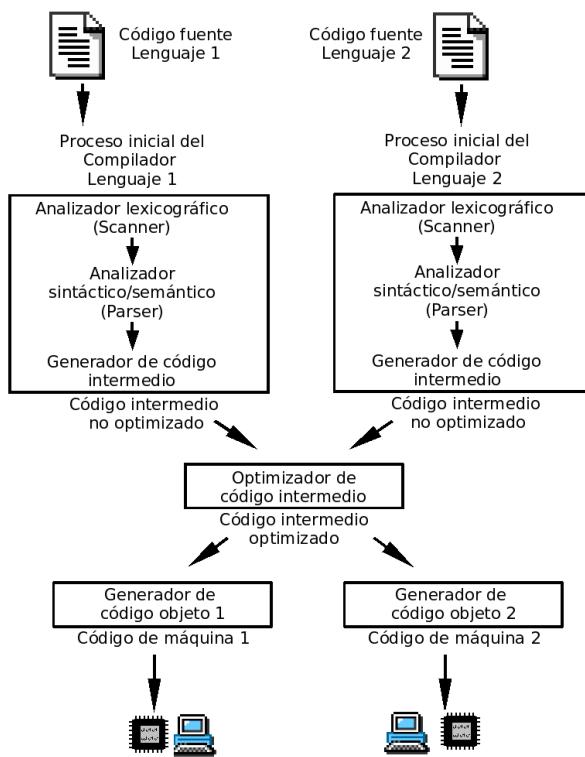
- ✓ Son mucho más fáciles de aprender y de utilizar que sus predecesores.
- ✓ Se reduce el tiempo para desarrollar programas, así como los costes.
- ✓ Son independientes del hardware, los programas pueden ejecutarse en diferentes tipos de máquina.
- ✓ La lectura, interpretación y modificación de los programas es mucho más sencilla.

Como desventajas se podría decir que el código objeto generado es menos eficiente que el código generado en lenguaje ensamblador, a pesar de que los compiladores realizan procesos de optimización del código.

Pero un programa que está escrito en un lenguaje de alto nivel también tiene que traducirse a un código que pueda utilizar la máquina. Los programas traductores que pueden realizar esta operación se llaman compiladores.

**Compilador:** Es un programa cuya función consiste en traducir el código fuente de un programa escrito en un lenguaje de alto nivel a lenguaje máquina. Al proceso de traducción se le conoce con el nombre de compilación.

Para ilustrar el proceso de compilación de programas observa la siguiente imagen:



Magnus Colossus-commonswiki (talk | contribs) (CC BY-SA)

El compilador realizará la traducción y además informará de los posibles errores. Una vez subsanados, se generará el programa traducido a código máquina, conocido como **código objeto**. Este programa aún no podrá ser ejecutado hasta que no se le añadan los módulos de enlace o bibliotecas, durante el proceso de enlazado. Una vez finalizado el enlazado, se obtiene el **código ejecutable**.

## Autoevaluación

Durante la fase de enlazado, se incluyen en el código fuente determinados módulos (bibliotecas) que son necesarios para que el programa pueda realizar ciertas tareas, posteriormente se obtendrá el código ejecutable.

- Verdadero  Falso

**Falso**

El código fuente es traducido por el compilador, pero en la fase de enlazado los módulos son añadidos al código objeto; estos módulos permitirán al programa manejar dispositivos, comunicarse con otros elementos del sistema, etc.

## 6.4.- Lenguajes interpretados.

Se caracterizan por estar diseñados para que su ejecución se realice a través de un **intérprete**. Cada instrucción escrita en un lenguaje interpretado se analiza, traduce y ejecuta tras haber sido verificada. Una vez realizado el proceso por el intérprete, la instrucción se ejecuta, pero no se guarda en memoria.

**Intérprete:** Es un programa traductor de un lenguaje de alto nivel en el que el proceso de traducción y de ejecución se llevan a cabo simultáneamente, es decir, la instrucción se pasa a lenguaje máquina y se ejecuta directamente. No se genera programa objeto, ni programa ejecutable.

Los lenguajes interpretados generan programas de menor tamaño que los generados por un compilador, al no guardar el programa traducido a código máquina. Pero presentan el inconveniente de ser algo más lentos, ya que han de ser traducidos durante su ejecución. Por otra parte, necesitan disponer en la máquina del programa intérprete ejecutándose, algo que no es necesario en el caso de un programa compilado, para los que sólo es necesario tener el programa ejecutable para poder utilizarlo.

Ejemplos de lenguajes interpretados son: **Perl, PHP, Python, JavaScript, etc.**



Stockbyte (CD-DVD) Num. V43 ([CC BY-NC](#))

A medio camino entre los lenguajes compilados y los interpretados, existen los lenguajes que podemos denominar **pseudo-compilados o pseudo-interpretados**, es el caso del **Lenguaje Java**. Java puede verse como compilado e interpretado a la vez, ya que su código fuente se compila para obtener el código binario en forma de bytecodes, que son estructuras parecidas a las instrucciones máquina, con la importante propiedad de no ser dependientes de ningún tipo de máquina (se detallarán más adelante). Los ficheros que contienen los bytecodes de Java tienen extensión .class. La Máquina Virtual Java se encargará de interpretar este código y, para su ejecución, lo traducirá a código máquina del procesador en particular sobre el que se esté trabajando.

## Autoevaluación

En Java el código fuente es compilado, obteniéndose el código binario en forma de bytecodes. Pero, ¿Cuál es la extensión del archivo resultante?

- Extensión `.obj`.
- Extensión `.class`.
- Extensión `.Java`.

Incorrecto, los archivos `.obj` son generados por un compilador antes de generar el archivo ejecutable.

Correcto, este tipo de archivos son los que la Máquina Virtual Java traducirá para poder ejecutarlos en la máquina real.

Incorrecto, los archivos con esta extensión contienen el código fuente del programa.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

## 7.- El lenguaje de programación Java.

### Caso práctico

**Ada** indica a **Juan** y **María** que el lenguaje elegido para sus desarrollos va a ser Java. La flexibilidad, facilidad de aprendizaje, similitud con algunos lenguajes que ya conocen y su capacidad para adaptarse a cualquier plataforma, hacen que sea ideal para producir las nuevas aplicaciones de BK Programación.



Stockbyte Num. ECD001 ([CC BY-NC](#))

## 7.1.- ¿Qué y cómo es Java?

Java es un lenguaje sencillo de aprender, con una sintaxis parecida a la de C++, pero en la que se han eliminado elementos complicados y que pueden originar errores. Java es orientado a objetos, con lo que elimina muchas preocupaciones al programador y permite la utilización de gran cantidad de bibliotecas ya definidas, evitando reescribir código que ya existe. Es un lenguaje de programación creado para satisfacer nuevas necesidades que los lenguajes existentes hasta el momento no eran capaces de solventar.

Una de las principales virtudes de Java es su independencia del hardware, ya que el código que se genera es válido para cualquier plataforma. Este código será ejecutado sobre una máquina virtual denominada **Maquina Virtual Java** (MVJ o JVM – Java Virtual Machine), que interpretará el código convirtiéndolo a código específico de la plataforma que lo soporta. De este modo el programa se escribe una única vez y puede hacerse funcionar en cualquier lugar. Lema del lenguaje: **“Write once, run everywhere”**.



[Robpatrick \(CC BY-NC-SA\)](#)

Antes de que apareciera Java, el lenguaje C era uno de los más extendidos por su versatilidad. Pero cuando los programas escritos en C aumentaban de volumen, su manejo comenzaba a complicarse. Mediante las técnicas de programación estructurada y programación modular se conseguían reducir estas complicaciones, pero no era suficiente.

Fue entonces cuando la Programación Orientada a Objetos (POO) entra en escena, aproximando notablemente la construcción de programas al pensamiento humano y haciendo más sencillo todo el proceso. Los problemas se dividen en objetos que tienen propiedades e interactúan con otros objetos, de este modo, el programador puede centrarse en cada objeto para programar internamente los elementos y funciones que lo componen.

Las características principales de lenguaje Java se resumen a continuación:

- ✓ El código generado por el compilador Java es independiente de la arquitectura.
- ✓ Está totalmente orientado a objetos.
- ✓ Su sintaxis es similar a C y C++.
- ✓ Es distribuido, preparado para aplicaciones TCP/IP.
- ✓ Dispone de un amplio conjunto de bibliotecas.
- ✓ Es robusto, realizando comprobaciones del código en tiempo de compilación y de ejecución.
- ✓ La seguridad está garantizada, ya que las aplicaciones Java no acceden a zonas delicadas de memoria o de sistema.

### Debes conocer

Obtén una descripción detallada de las características reseñadas anteriormente a través del siguiente artículo:

[Características detalladas del lenguaje Java](#)

Java es uno de los lenguajes más utilizados en la actualidad, sobre todo para aplicaciones de Internet.

## 7.2.- Breve historia.

Java surgió en 1991 cuando un grupo de ingenieros de Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a programar pequeños dispositivos electrónicos. La dificultad de estos dispositivos es que cambian continuamente y para que un programa funcione en el siguiente dispositivo aparecido, hay que rescribir el código. Por eso la empresa Sun quería crear un lenguaje **independiente del dispositivo**.

Pero no fue hasta 1995 cuando pasó a llamarse **Java**, dándose a conocer al público como lenguaje de programación para computadores. Java pasa a ser un lenguaje totalmente independiente de la plataforma y a la vez potente y orientado a objetos. Esta filosofía y su facilidad para crear aplicaciones para redes TCP/IP ha hecho que sea uno de los lenguajes más utilizados en la actualidad.

El factor determinante para su expansión fue la incorporación de un intérprete Java en la versión 2.0 del navegador Web Netscape Navigator, lo que supuso una gran revuelo en Internet. A principios de 1997 apareció **Java 1.1** que proporcionó sustanciales mejoras al lenguaje. **Java 1.2**, más tarde rebautizado como **Java 2**, nació a finales de 1998.

El principal objetivo del lenguaje Java es llegar a ser el **nexo universal** que conecte a los usuarios con la información, ésta situada en el ordenador local, en un servidor Web, en una base de datos o en cualquier otro lugar.

Para el desarrollo de programas en lenguaje Java es necesario utilizar un entorno de desarrollo denominado **JDK** (Java Development Kit), que provee de un compilador y un entorno de ejecución (**JRE** – Java Run Environment) para los bytecodes generados a partir del código fuente. Al igual que las diferentes versiones del lenguaje han incorporado mejoras, el entorno de desarrollo y ejecución también ha sido mejorado sucesivamente.

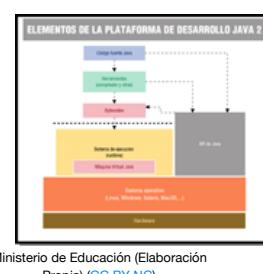
**Java 2** es la tercera versión del lenguaje, pero es algo más que un lenguaje de programación, incluye los siguientes elementos:

- ✓ Un lenguaje de programación: Java.
- ✓ Un conjunto de bibliotecas estándar que vienen incluidas en la plataforma y que son necesarias en todo entorno Java. Es el Java Core.
- ✓ Un conjunto de herramientas para el desarrollo de programas, como es el compilador de bytecodes, el generador de documentación, un depurador, etc.
- ✓ Un entorno de ejecución que en definitiva es una máquina virtual que ejecuta los programas traducidos a bytecodes.

El siguiente esquema muestra los elementos fundamentales de la plataforma de desarrollo Java 2.

La plataforma Java 2 no para de crecer debido a su amplio uso, añadiendo continuamente tecnologías que aportan nuevas funcionalidades. Entre otras, actualmente existen:

- ✓ **J2SE**: Entorno de Sun relacionado con la creación de aplicaciones y applets en lenguaje Java para su ejecución en equipo y servidores.
- ✓ **J2EE**: Pensada para la creación de aplicaciones web Java empresariales y del lado del servidor.
- ✓ **Java FX**: Permite crear e implementar Aplicaciones de Internet Enriquecidas (RIAs), cuya interfaz se comparte igual en distintas plataformas.
- ✓ **Java Embedded**: Plataforma creada para su ejecución en sistemas embebidos. Está muy relacionada con el conocido Internet of Things (Internet de las cosas).
- ✓ **Java Card**: tecnología que permite la ejecución de pequeñas aplicaciones Java en tarjetas inteligentes.



Ministerio de Educación (Elaboración Propia) [\(CC BY-NC\)](https://creativecommons.org/licenses/by-nc/4.0/)

### Para saber más

Si deseas conocer más sobre la historia del lenguaje Java, aquí te ofrecemos más información:

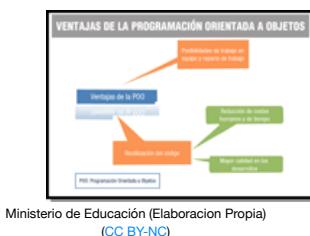
[Historia de Java](#)

## 7.3.- La POO y Java.

En Java, los datos y el código (funciones o métodos) se combinan en entidades llamadas **objetos**. El objeto tendrá un comportamiento (su código interno) y un estado (los datos). Los objetos permiten la reutilización del código y pueden considerarse, en sí mismos, como piezas reutilizables en múltiples proyectos distintos. Esta característica permite reducir el tiempo de desarrollo de software.

Por simplificar un poco las cosas, un programa en Java será como una representación teatral en la que debemos preparar primero cada personaje, definir sus características y qué va a saber hacer. Cuando esta fase esté terminada, la obra se desarrollará sacando personajes a escena y haciéndoles interactuar.

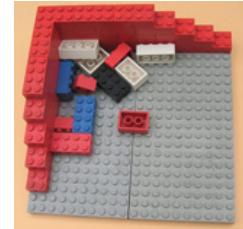
Al emplear los conceptos de la Programación Orientada a Objetos (POO), Java incorpora las tres características propias de este paradigma: **encapsulación**, **herencia** y **polimorfismo**. Los patrones o tipos de objetos se denominan **clases** y los objetos que utilizan estos patrones o pertenecen a dichos tipos, se identifican con el nombre de **instancias**. Pero, no hay que alarmarse, estos conceptos se verán más adelante en sucesivas unidades.



Ministerio de Educación (Elaboración Propia)

(CC BY-NC)

Otro ejemplo para seguir aclarando ideas, piensa en los bloques de juegos de construcción. Suponemos que conoces los cubos de plástico en varios colores y tamaños. Por una de sus caras disponen de pequeños conectores circulares y en otra de sus caras pequeños orificios en los que pueden conectarse otros bloques, con el objetivo principal de permitir construir formas más grandes. Si usas diferentes piezas del lego puedes construir aviones, coches, edificios, etc. Si te fijas bien, cada pieza es un objeto pequeño que puede unirse con otros objetos para crear objetos más grandes.



Priwo (Dominio público)

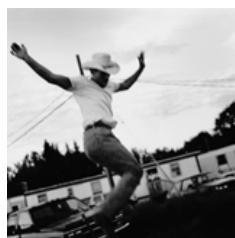
Pues bien, aproximadamente así es como funciona la programación dirigida a objetos: unimos elementos pequeños para construir otros más grandes. Nuestros programas estarán formados por muchos componentes (objetos) independientes y diferentes; cada uno con una función determinada en nuestro software y que podrá comunicarse con los demás de una manera predefinida.

## 7.4.- Independencia de la plataforma y trabajo en red.

Existen dos características que distinguen a **Java** de otros lenguajes, como son la **independencia de la plataforma** y la posibilidad de trabajar en red o, mejor, la posibilidad de **crear aplicaciones que trabajan en red**.

Estas características las vamos a explicar a continuación:

- a. **Independencia:** Los programas escritos en Java pueden ser ejecutados en cualquier tipo de hardware. El código fuente es compilado, generándose el código conocido como **Java Bytecode** (instrucciones máquina simplificadas que son específicas de la plataforma Java), el bytecode será interpretado y ejecutado en la **Máquina Virtual Java (MVJ o JVM – Java Virtual Machine)** que es un programa escrito en código nativo de la plataforma destino entendible por el hardware. Con esto se evita tener que realizar un programa diferente para cada CPU o plataforma. Por tanto, la parte que realmente es dependiente del sistema es la Máquina Virtual Java, así como las librerías o bibliotecas básicas que permiten acceder directamente al hardware de la máquina.
- b. **Trabajo en red:** Esta capacidad del lenguaje ofrece múltiples posibilidades para la comunicación vía TCP/IP. Para poder hacerlo existen librerías que permiten el acceso y la interacción con protocolos como ..... http, ..... ftp, etc., facilitando al programador las tareas del tratamiento de la información a través de redes.



Stockbyte (DVD-CD) Num. V43 ([CC BY-NC](#))



Stockbyte (DVD-CD) Num. 109 ([CC BY-NC](#))

## Autoevaluación

¿Qué elemento es imprescindible para que una aplicación escrita en Java pueda ejecutarse en un ordenador?

- Que disponga de conexión a Internet y del hardware adecuado.
- Que tenga instalado un navegador web y conexión a Internet.
- Que tenga la Máquina Virtual Java adecuada instalada.

Estos elementos no son suficientes para poder hacer funcionar una aplicación escrita en lenguaje Java.

Tener conectividad a Internet no es imprescindible para poder ejecutar programas Java, además de que no sólo a través del navegador puede ejecutarse código Java.

Efectivamente, sin la Máquina Virtual Java es imposible que el hardware pueda entender los códigos de bytes necesarios para la ejecución del programa, siendo necesaria la máquina virtual adecuada para la plataforma hardware que estemos utilizando.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

## 7.5.- Seguridad y simplicidad.

Junto a las características diferenciadoras del lenguaje Java relacionadas con la independencia y el trabajo en red, han de destacarse dos virtudes que hacen a este lenguaje uno de los más extendidos entre la comunidad de programadores: su seguridad y su simplicidad.

- a. **Seguridad:** En primer lugar, los posibles accesos a zonas de memoria “sensibles” que en otros lenguajes como C y C++ podían suponer peligros importantes, se han eliminado en Java.



Stockbyte (DVD-CD) Num. 109 ([CC BY-NC](#))

En segundo lugar, el código Java es comprobado y verificado para evitar que determinadas secciones del código produzcan efectos no deseados. Los test que se aplican garantizan que las operaciones, operandos, conversiones, uso de clases y demás acciones son seguras.

Y en tercer lugar, Java no permite la apertura de ficheros en la máquina local, tampoco permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra.

En definitiva, podemos afirmar que Java es un lenguaje seguro.

- b. **Simplicidad:** Aunque Java es tan potente como C o C++, es bastante más sencillo. Posee una curva de aprendizaje muy rápida y, para alguien que comienza a programar en este lenguaje, le resulta relativamente fácil comenzar a escribir aplicaciones interesantes.

Si has programado alguna vez en C o C++ encontrarás que Java te pone las cosas más fáciles, ya que se han eliminado: la aritmética de ..... punteros, los registros, la definición de tipos, la gestión de memoria, etc. Con esta simplificación se reduce bastante la posibilidad de cometer errores comunes en los programas. Un programador experimentado en C o C++ puede cambiar a este lenguaje rápidamente y obtener resultados en muy poco espacio de tiempo.

Muy relacionado con la simplicidad que aporta Java está la incorporación de un elemento muy útil como es el **Recolector de Basura (Garbage collector)**. Permite al programador liberarse de la gestión de la memoria y hace que ciertos bloques de memoria puedan reaprovecharse, disminuyendo el número de huecos libres (..... fragmentación de memoria).

Cuando realicemos programas, crearemos objetos, haremos que éstos interaccionen, etc. Todas estas operaciones requieren de uso de memoria del sistema, pero la gestión de ésta será realizada de manera transparente al programador. Todo lo contrario que ocurría en otros lenguajes. Podremos crear tantos objetos como solicitemos, pero nunca tendremos que destruirlos. El entorno de Java borrará los objetos cuando determine que no se van a utilizar más. Este proceso es conocido como recolección de basura.

### Autoevaluación

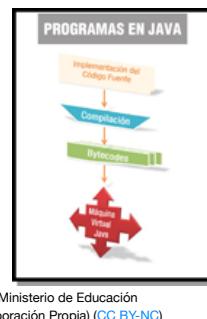
#### Rellena los huecos con los conceptos adecuados:

En Java se ha simplificado la gestión de memoria a través de la eliminación de la Aritmética de ..... , por lo que la incorporación del Garbage Collector evita que se produzca un crecimiento de los huecos libres en memoria, que recibe el nombre de ..... de memoria.

Los Punteros son un tipo especial de elemento utilizado en C/C++ que permiten realizar directamente gestión de la memoria del sistema, su control es complicado y, en ocasiones, peligroso. Al dejar en manos del recolector de basura la gestión de la memoria, se evita la Fragmentación y se reutilizan mejor los espacios libres. Liberando al programador para que se centre en el desarrollo del programa sin distracciones adicionales.

## 7.6.- Java y los Bytecodes.

Un programa escrito en Java no es directamente ejecutable, es necesario que el código fuente sea interpretado por la Maquina Virtual Java. ¿Cuáles son los pasos que se siguen desde que se genera el código fuente hasta que se ejecuta? A continuación se detallan cada uno de ellos.



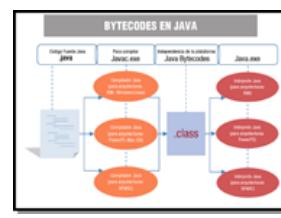
1. Una vez escrito el código fuente (archivos con extensión .Java), éste es precompilado generándose los códigos de bytes, Bytecodes o Java Bytecodes (archivos con extensión .class)
2. Los archivos de bytecodes serán interpretados directamente por la Maquina Virtual Java y traducidos a código nativo de la plataforma sobre la que se esté ejecutando el programa.

**Bytecode:** Son un conjunto de instrucciones en lenguaje máquina que no son específicas a ningún procesador o sistema de cómputo. Un intérprete de código de bytes (bytecodes) para una plataforma específica será quien los ejecute. A estos intérpretes también se les conoce como Máquinas Virtuales Java o intérpretes Java de tiempo de ejecución.

En el proceso de precompilación, existe un verificador de códigos de bytes que se asegurará de que se cumplen las siguientes condiciones:

- ✓ El código satisface las especificaciones de la Máquina Virtual Java.
- ✓ No existe amenaza contra la integridad del sistema.
- ✓ No se producen desbordamientos de memoria.
- ✓ Los parámetros y sus tipos son adecuados.
- ✓ No existen conversiones de datos no permitidas.

Para que un bytecode pueda ser ejecutado en cualquier plataforma, es imprescindible que dicha plataforma cuente con el intérprete adecuado, es decir, la máquina virtual específica para esa plataforma. En general, la Máquina Virtual Java es un programa de reducido tamaño y gratuito para todos los sistemas operativos.



## 8.- Programas en Java.

### Caso práctico

**Juan** celebra que BK Programación vaya a desarrollar sus programas en un lenguaje como Java. En algunas ocasiones ha asistido a congresos y ferias de exposiciones de software en las que ha podido intercambiar impresiones con compañeros de profesión sobre los diferentes lenguajes que utilizan en sus proyectos. Una gran mayoría destacaba lo fácil y potente que es programar en Java.

**Juan** está entusiasmado y pregunta: —**Ada**, cuándo empezamos? ¿Tienes código fuente para empezar a ver la sintaxis? ¿Podremos utilizar algún entorno de desarrollo profesional?

**Ada** responde sonriendo: —¡Manos a la obra! María, ¿preparada? Vamos a echarle un vistazo a este fragmento de código...



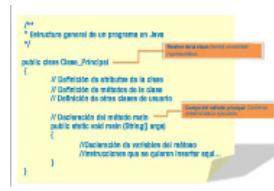
Ziko van Dijk (CC BY-SA)

Hasta ahora, hemos descrito el lenguaje de programación Java, hemos hecho un recorrido por su historia y nos hemos instruido sobre su filosofía de trabajo, pero te preguntarás ¿Cuándo empezamos a desarrollar programas? ¿Qué elementos forman parte de un programa en Java? ¿Qué se necesita para programar en este lenguaje? ¿Podemos crear programas de diferente tipo?

No te impacientes, cada vez estamos más cerca de comenzar la experiencia con el lenguaje de programación Java. Iniciaremos nuestro camino conociendo cuales son los elementos básicos de un programa Java, la forma en que debemos escribir el código y los tipos de aplicaciones que pueden crearse en este lenguaje.

## 8.1.- Estructura de un programa.

En el gráfico al que puedes acceder a continuación, se presenta la estructura general de un programa realizado en un lenguaje orientado a objetos como es Java.



Jose Luis García Martínez (Elaboración propia)

[CC BY-NC](#)

Vamos a analizar cada uno de los elementos que aparecen en dicho gráfico:

**public class Clase\_Principal:** Todos los programas han de incluir una clase como ésta. Es una clase general en la que se incluyen todos los demás elementos del programa. Entre otras cosas, contiene el método o función `main()` que representa al programa principal, desde el que se llevará a cabo la ejecución del programa. Este método es el que ejecutará en primer lugar cuando una aplicación se lance a ejecución. Esta clase puede contener a su vez otras clases del usuario, pero sólo una puede ser `public`. El nombre del fichero `Java` que contiene el código fuente de nuestro programa, coincidirá con el nombre de la clase que estamos describiendo en estas líneas.

### Recomendación

Ten en cuenta que **Java distingue entre mayúsculas y minúsculas**. Si le das a la clase principal el nombre `PrimerPrograma`, el archivo `Java` tendrá como identificador `PrimerPrograma.Java`, que es totalmente diferente a `primerprograma.Java`. Además, para Java los elementos `PrimerPrograma` y `primerprograma` serían considerados dos clases diferentes dentro del código fuente.

- ✓ **public static void main (String[] args):** Es el método que representa al programa principal, en él se podrán incluir las instrucciones que estimemos oportunas para la ejecución del programa. Desde él se podrá hacer uso del resto de clases creadas. Todos los programas Java tienen un método `main`.
- ✓ **Comentarios:** Los comentarios se suelen incluir en el código fuente para realizar aclaraciones, anotaciones o cualquier otra indicación que el programador estime oportuna. Estos comentarios pueden introducirse de dos formas, `con // y con /* */`. Con la primera forma estaríamos estableciendo una única línea completa de comentario y, con la segunda, con `/*` comenzaríamos el comentario y éste no terminaría hasta que no insertáramos `*/`.
- ✓ **Bloques de código:** son conjuntos de instrucciones que se marcan mediante la apertura y cierre de llaves `{ }`. El código así marcado es considerado interno al bloque.
- ✓ **Punto y coma:** aunque en el ejemplo no hemos incluido ninguna línea de código que termine con punto y coma, hay que hacer hincapié en que cada línea de código ha de terminar con punto y coma `(;)`. En caso de no hacerlo, tendremos errores sintácticos.

### Autoevaluación

`public static void main (String[] args)` es la clase general del programa.

Verdadero  Falso

Falso

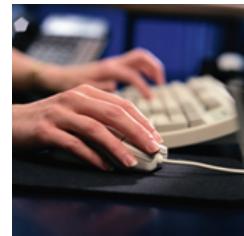
La clase general del programa tiene el formato `public class <nombre_clase_general>` y todos los programas Java

tendrán una. Dentro de ella podrá haber más clases definidas por el usuario y siempre, deberá haber un método main desde el que se irá haciendo uso del resto de clases definidas.

## 8.2.- El entorno básico de desarrollo Java.

Ya conoces cómo es la estructura de un programa en Java, pero, ¿qué necesitamos para llevarlo a la práctica? La herramienta básica para empezar a desarrollar aplicaciones en Java es el **JDK (Java Development Kit o Kit de Desarrollo Java)**, que incluye un compilador y un intérprete para línea de comandos. Estos dos programas son los empleados en la precompilación e interpretación del código.

Como veremos, existen diferentes entornos para la creación de programas en Java que incluyen multitud de herramientas, pero por ahora nos centraremos en el entorno más básico, extendido y gratuito, el Java Development Kit (JDK).



Stockbyte (DVD-CD) Num. V43 ([CC BY-NC](#))

Según se indica en la propia página web de Oracle, JDK es un entorno de desarrollo para construir aplicaciones, applets y componentes utilizando el lenguaje de programación Java. Incluye herramientas útiles para el desarrollo y prueba de programas escritos en Java y ejecutados en la Plataforma Java.

Así mismo, junto a la JDK se incluye una implementación del entorno de ejecución Java, el **JRE (Java Runtime Environment)** para ser utilizado por el JDK.

El JRE incluye la Máquina Virtual de Java (MVJ ó JVM – Java Virtual Machine), bibliotecas de clases y otros ficheros que soportan la ejecución de programas escritos en el lenguaje de programación Java.

### Debes conocer

Para poder utilizar JDK y JRE es necesario realizar la descarga e instalación. Instalaremos la última versión estable del JDK: OracleJDK 14. Puedes seguir los pasos del proceso a continuación:

◀ 1 2 3 4 5 ▶

### Acceso a la web de descarga

1. Accedemos a la web de descarga de oracle: [Web de Descarga de Oracle](#). También puedes optar por descargar la versión totalmente libre del JDK: OpenJDK: [Web de Descarga de OpenJDK](#). Recuerda que ambas versiones tienen la misma funcionalidad, la diferencia radicará en el uso que daremos a las aplicaciones que desarrollaremos.

The screenshot shows the Oracle Java SE Downloads page. At the top, there is a navigation bar with links for Products, Resources, Support, and View Accounts. Below the navigation bar, the page title is "Java SE Downloads" and the subtitle is "Java Platform, Standard Edition". A sub-section titled "Java SE 14" is displayed, stating "Java SE 14.0.2 is the latest release for the Java SE Platform". To the left of the main content area, there is a sidebar with links for Documentation, Installation Instructions, Release Notes, Oracle License (with sub-links for Binary License and Documentation License), Java SE Licensing Information User Manual (with sub-links for Includes Third Party Licenses and Certified System Configurations), and Readme. To the right of the main content area, there is a section titled "Oracle JDK" with two download links: "JDK Download" and "Documentation Download".

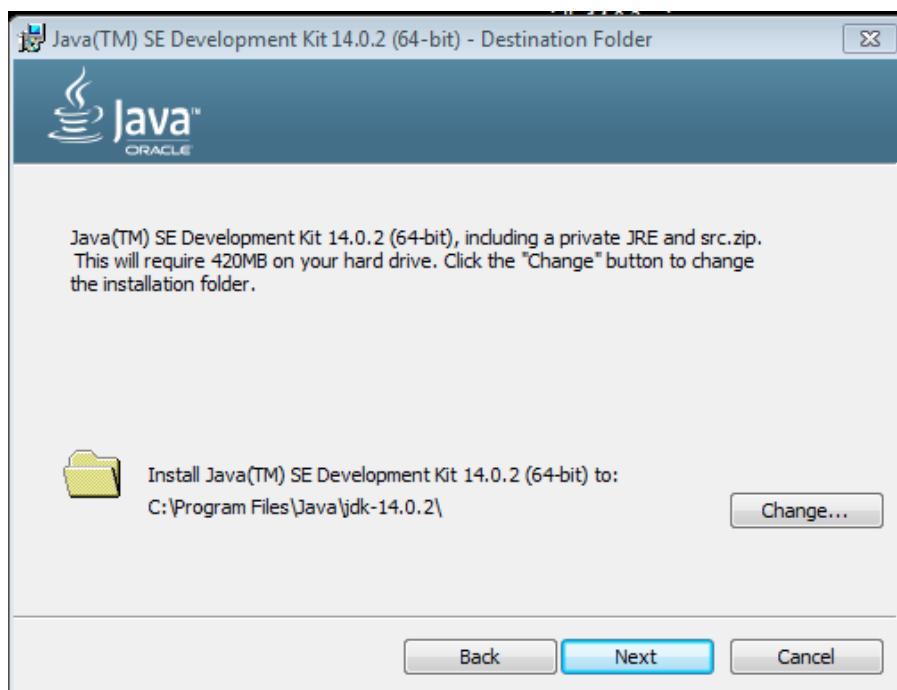
## Descarga de la versión

1. Debemos descargar la versión adecuada según el sistema operativo a utilizar. Es posible descargar el instalador o un archivo comprimido .zip. Mejor el archivo ejecutable.

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE		
Product / File Description	File Size	Download
Linux Debian Package	157.92 MB	 jdk-14.0.1_linux-x64_bin.deb
Linux RPM Package	165.04 MB	 jdk-14.0.1_linux-x64_bin.rpm
Linux Compressed Archive	182.04 MB	 jdk-14.0.1_linux-x64_bin.tar.gz
macOS Installer	175.77 MB	 jdk-14.0.1_osx-x64_bin.dmg
macOS Compressed Archive	176.19 MB	 jdk-14.0.1_osx-x64_bin.tar.gz
Windows x64 Installer	162.07 MB	 jdk-14.0.1_windows-x64_bin.exe
Windows x64 Compressed Archive	181.53 MB	 jdk-14.0.1_windows-x64_bin.zip

## Instalación del JDK

1. Una vez descargado el fichero ejecutable, tan solo tenemos que hacer doble click para iniciar la instalación. Una serie de ventajas no guiarán sobre el proceso, que avanzará a golpe de ratón.
2. Observa en uno de los pasos el directorio (en este caso en Windows) donde se hará la instalación de las herramientas. Es importante conocer su ubicación.



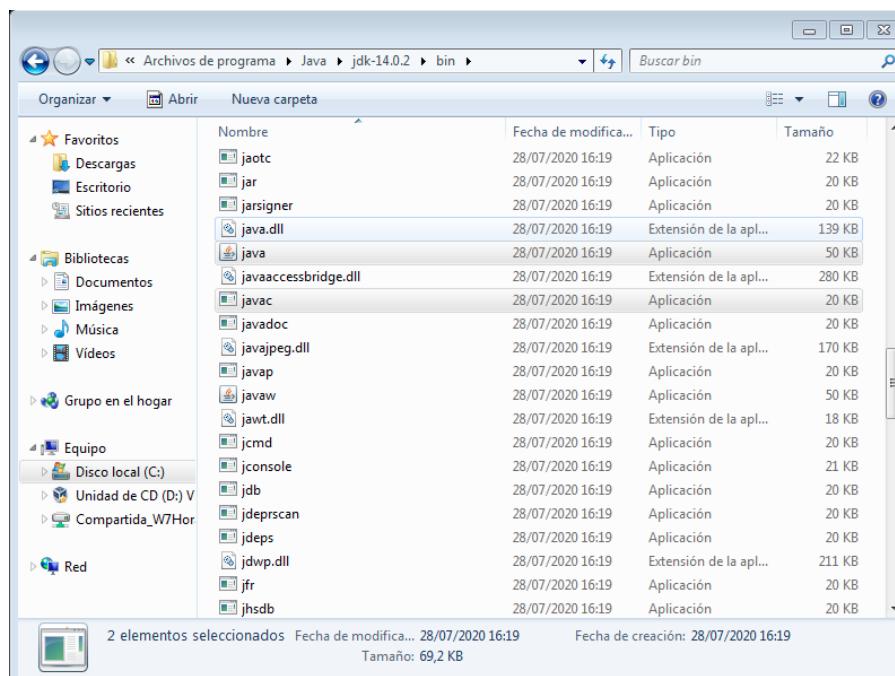
Pasados unos minutos, la instalación habrá finalizado.

## Acceso al directorio de instalación del JDK

1. Podemos comprobar que en la ruta indicada en el paso anterior se encuentran todos los

ficheros que forman el JDK.

2. Observa los dos ficheros seleccionados en la imagen, los utilizaremos en breve (**java** para lanzar a ejecución una aplicación y **javac** para compilar).



## Acceso al directorio de instalación del JDK

Todas las imágenes utilizadas con propiedad del Ministerio de Educación y FP bajo licencia CC-BY-NC se corresponden con capturas de pantalla.

[Resumen textual alternativo](#)

Para poder desarrollar nuestros primeros programas en Java sólo necesitaremos un editor de texto plano y algunas de las herramientas que acabamos de instalar.

## Para saber más

Hasta la versión 11 del JDK, todas las herramientas podían descargarse y utilizarse para programar sin ningún tipo de limitación. Además, las aplicaciones desarrolladas podían ponerse en producción o ser distribuidas sin pagar ningún tipo de licencia. Sin embargo, a partir de la versión 11 se pueden utilizar todas las herramientas de desarrollo sin limitación, pero tendremos que pagar una licencia a Oracle si queremos poner nuestras aplicaciones desarrolladas con OracleJDK en producción. Esto no afecta a versiones anteriores del JDK.

Si queremos desarrollar aplicaciones Java que pondremos en producción en un futuro, tendremos que tener en cuenta algunos aspectos. En el siguiente enlace tienes información al respecto.

[Licencia de OracleJDK](#)

Como se comenta en el artículo, la alternativa totalmente gratuita es el OpenJDK. En el siguiente enlace puedes comprobar las diferencias entre ambas plataformas.

[Diferencias entre OpenJDK y OracleJDK](#)

¿Utilizas como sistema operativo Linux?

En el siguiente enlace tienes los pasos para realizar la instalación del OpenJDK y OracleJDK en Ubuntu.

[Instalación de JDK en Ubuntu](#)

## Autoevaluación

**Podemos desarrollar programas escritos en Java mediante un editor de textos y a través del JRE podemos ejecutarlos.**

- Verdadero  Falso

**Verdadero**

Efectivamente, JRE incluye un subconjunto de JDK que permitiría realizar la compilación del código fuente y la ejecución posterior en la Máquina Virtual Java de nuestro programa.

## 8.3.- La API de Java.

Junto con el kit de desarrollo que hemos descargado e instalado anteriormente, vienen incluidas gratuitamente todas las bibliotecas de la API (Application Programming Interface – Interfaz de programación de aplicaciones) de Java, es lo que se conoce como Bibliotecas de Clases Java. Este conjunto de bibliotecas proporciona al programador paquetes de clases útiles para la realización de múltiples tareas dentro de un programa. Está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.

En décadas pasadas una biblioteca era un conjunto de programas que contenían cientos de rutinas (una rutina es un procedimiento o función bien verificados, en determinado lenguaje de programación). Las rutinas de biblioteca manejaban las tareas que todos o casi todos los programas necesitaban. El programador podía recurrir a esta biblioteca para desarrollar programas con rapidez.



Stockbyte (DVD-CD) Num. SD174 [CC BY-NC](#)

Una biblioteca de clases es un conjunto de clases de programación orientada a objetos. Esas clases contienen métodos que son útiles para los programadores. En el caso de Java cuando descargamos el JDK obtenemos la biblioteca de clases API. Utilizar las clases y métodos de las APIs de Java reduce el tiempo de desarrollo de los programas. También, existen diversas bibliotecas de clases desarrolladas por terceros que contienen componentes reutilizables de software, y están disponibles a través de la Web.

### Para saber más

Si quieres acceder a la información oficial sobre la API de Java, te proponemos el siguiente enlace (está en Inglés).

[Información oficial sobre la API de Java](#)

### Autoevaluación

**Indica qué no es la API de Java:**

- Un entorno integrado de desarrollo.
- Un conjunto de bibliotecas de clases.
- Una parte del JDK, incluido en el Java SE.

En efecto, la API de Java es utilizada para la creación de programas pero no ofrece las herramientas de desarrollo que un IDE posee.

Incorrecto, la API de Java provee de clases agrupadas en paquetes que proporcionan una interfaz común para desarrollar aplicaciones Java en todas las plataformas.

Incorrecto, la API de Java está incluida junto con JDK y JRE en Java SE.

## **Solución**

1. Opción correcta
2. Incorrecto
3. Incorrecto

## 8.4.- Afinando la configuración.

Para que podamos compilar y ejecutar ficheros Java es necesario que realicemos unos pequeños ajustes en la configuración del sistema. Vamos a indicarle dónde encontrar los ficheros necesarios para realizar las labores de compilación y ejecución, en este caso `Javac.exe` y `Java.exe`, así como las librerías contenidas en la API de Java y las clases del usuario.

**La variable PATH:** Como aún no disponemos de un IDE (Integrated Development Environment - Entorno Integrado de Desarrollo) la única forma de ejecutar programas es a través de línea de comandos. Pero sólo podremos ejecutar programas directamente si la ruta hacia ellos está indicada en la variable PATH del ordenador. Es necesario que incluyamos la ruta hacia estos programas en nuestra variable PATH. Esta ruta será el lugar donde se instaló el JDK hasta su directorio `bin`.

Para ello, sigue las indicaciones que te mostramos a continuación:



Stockbyte (DVD-CD) Num. EP006 ([CC BY-NC](#))

### Debes conocer

En la siguiente sección aprenderás como configurar la variable PATH en Windows.

◀ 1 2 3 4 5 6 7 8 ▶

### Ejecutar la aplicación java desde la consola

Una vez realizada la instalación, si abrimos una consola en Windows y ejecutamos el comando `java` podremos comprobar que no funciona: el problema es que Windows no sabe dónde está almacenado dicho comando. Obsérvalo en la siguiente imagen:

```
C:\>java
"java" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
C:\>
```

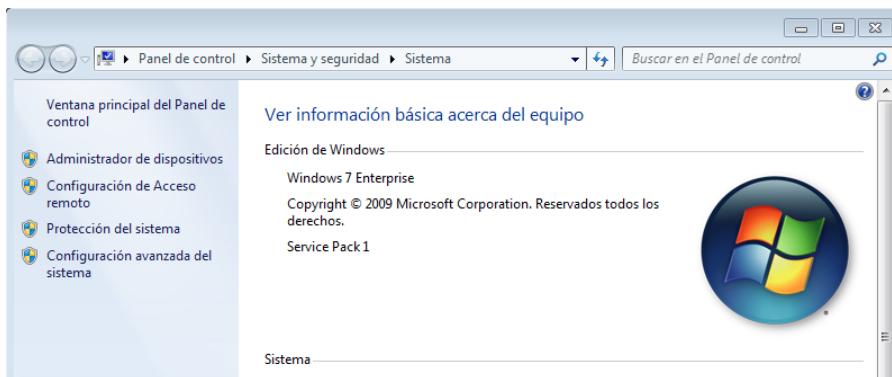
### Ejecución del comando PATH

Si ejecutas el comando `PATH`, podrás comprobar el contenido de la variable de entorno `PATH`. Solo los comandos incluidos en las carpetas que contiene la variable `PATH` podrán ser ejecutados directamente desde cualquier punto del sistema de archivos sin tener que indicar su ruta absoluta.

```
C:\>java  
"java" no se reconoce como un comando interno o externo,  
programa o archivo por lotes ejecutable.  
C:\>PATH  
PATH=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32  
\\WindowsPowerShell\v1.0\  
C:\>
```

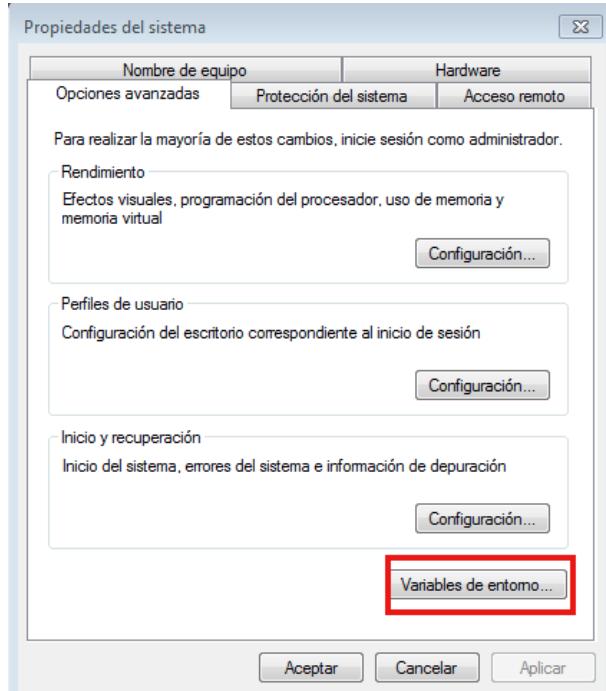
## Configurar la variable PATH

Para configurar la variable PATH accedemos al **Panel de Control - Sistema y Seguridad - Sistema - Configuración Avanzada del Sistema** (en Windows).



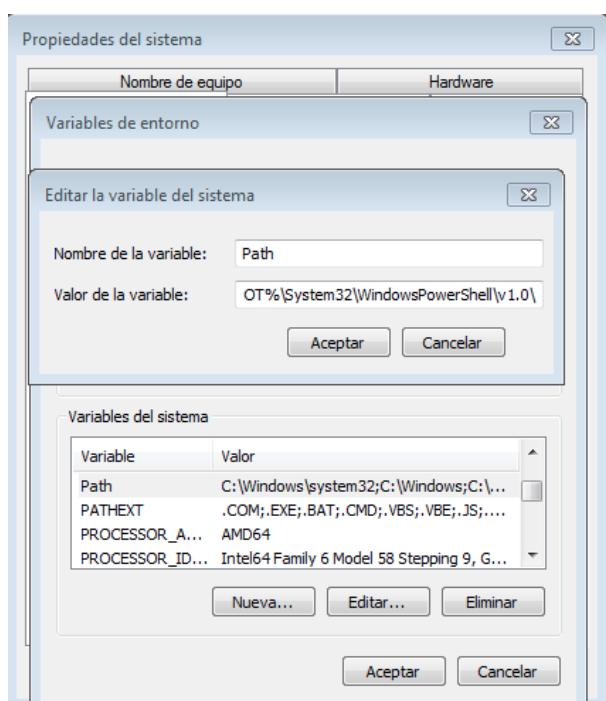
## Configurar la variable PATH II

En la pestaña **Opciones Avanzadas** debemos pulsar sobre **Variables de Entorno**.



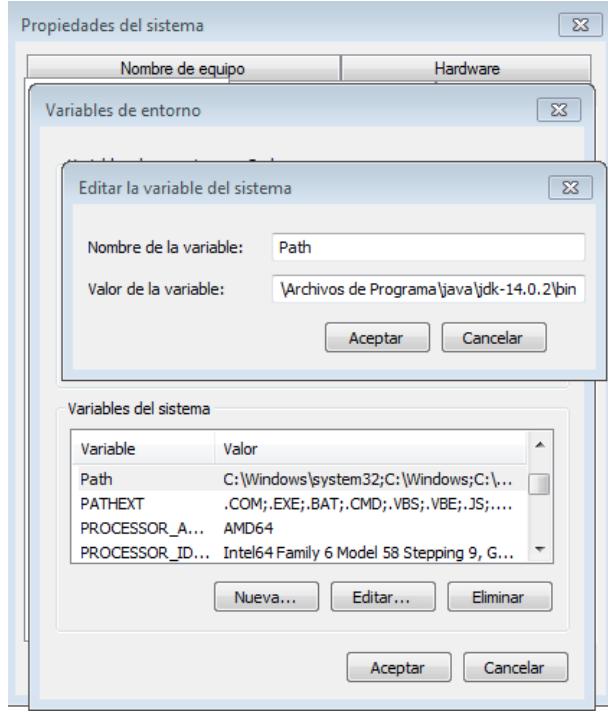
## Configurar la variable PATH III

En el panel **Variables del Sistema** debemos seleccionar la variable **PATH** y después pulsar sobre el botón **Editar**. Aparecerá una pequeña ventana donde modificar el variable de la variable.



## Configurar la variable PATH IV

Para terminar, añade al final de la variable PATH el símbolo ; seguido de la ruta de instalación del JDK, en concreto, de la carpeta **bin** (contiene los ejecutables).



## Configurar la variable PATH V

Ya puedes ejecutar los comandos de la jdk de java desde la consola, sin necesidad de indicar la ruta donde se encuentran. Obsérvalo en la siguiente imagen, donde se ejecuta el comando `java -version` para mostrar la versión del jdk instalada.

```
C:\>java -version
java version "14.0.2" 2020-02-14
Java(TM) SE Runtime Environment (build 14.0.2+12-46)
Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing)
C:\>
```

## Configurar la variable PATH VI

Todos las imágenes utilizadas con propiedad del Ministerio de Educación y FP bajo licencia CC-BY-NC y se corresponden con capturas de pantalla.

[Resumen textual alternativo](#)

## Para saber más

Si deseas conocer más sobre la configuración de variables de entorno en sistemas Windows y Linux, te proponemos el siguiente material:

## Configuración de la variable PATH en Ubuntu

[https://www.youtube.com/embed/lMux\\_rk8\\_I](https://www.youtube.com/embed/lMux_rk8_I)

Didier Almaraz, Configuración de la variable PATH en Windows (CC0)

**La variable CLASSPATH:** esta variable de entorno establece dónde buscar las clases o bibliotecas de la API de Java, así como las clases creadas por el usuario. Es decir, los ficheros .class que se obtienen una vez compilado el código fuente de un programa escrito en Java. Es posible que en dicha ruta existan directorios y ficheros comprimidos en los formatos zip o jar que pueden ser utilizados directamente por el JDK, conteniendo en su interior archivos con extensión class.

(Por ejemplo: C:\Program Files\Java\jdk-14.0\_2\bin)

Si no existe la variable **CLASSPATH** debes crearla, para modificar su contenido sigue el mismo método que hemos empleado para la modificación del valor de la variable **PATH**, anteriormente descrito. Ten en cuenta que la ruta que debes incluir será el lugar donde se instaló el JDK hasta su directorio lib.

(Por ejemplo: C:\Archivos de Programa\java\jdk-14.0.2\lib)

## Autoevaluación

**¿Qué variable de sistema o de entorno debemos configurar correctamente para que podamos compilar directamente desde la línea de comandos nuestros programas escritos en lenguaje Java?**

- CLASSPATH.**
- PATH.**
- Javac.exe.**

No es correcto, esta es la variable que hemos de configurar para conseguir que las clases de Java o las creadas por el usuario estén accesibles por nuestros programas.

Efectivamente, esta es la variable de entorno que modificaremos añadiendo a su contenido la ruta hasta el directorio bin donde está instalado el JDK.

No es correcto, la pregunta te pedía cuál es la variable, no cuál es el programa para realizar la compilación.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

## 8.5.- Codificación, compilación y ejecución de aplicaciones.

Una vez que la configuración del entorno Java está completada y tenemos el código fuente de nuestro programa escrito en un archivo con extensión `.Java`, la compilación de aplicaciones se realiza mediante el programa **Javac** incluido en el software de desarrollo de Java.

Para llevar a cabo la compilación desde la línea de comandos, escribiremos:

```
javac archivo.java
```

Donde `Javac` es el compilador de Java y `archivo.java` es nuestro código fuente.

Si el compilador detecta errores, los mostrará en el terminal. Como programadores, tendremos que corregirlos para poder generar el código objeto. Normalmente el compilador nos proporciona información sobre los errores detectados con el objetivo de facilitarnos su corrección.

El resultado de la compilación será un archivo con el mismo nombre que el archivo Java pero con la **extensión class**. Esto ya es el archivo con el código en forma de bytecode. Es decir con el código precompilado. Si en el código fuente de nuestro programa figuraran más de una clase, veremos como al realizar la compilación se generarán tantos archivos con extensión `.class` como clases tengamos. Además, si estas clases tenían método `main` podremos ejecutar dichos archivos por separado para ver el funcionamiento de dichas clases.

Para que el programa pueda ser ejecutado, siempre y cuando esté incluido en su interior el método `main`, podremos utilizar el intérprete incluido en el kit de desarrollo.



Stockbyte (DVD-CD) Num. EP006 ([CC0](#))

La ejecución de nuestro programa desde la línea de comandos podremos hacerla escribiendo:

```
java archivo
```

Donde `java` es el intérprete y `archivo` es el archivo con el código precompilado.

### Ejercicio resuelto

Vamos a llevar a la práctica todo lo que hemos estado detallando a través de la creación, compilación y ejecución de un programa sencillo escrito en Java.

Observa el código que se muestra más abajo, seguro que podrás entender parte de él. Cópialo en un editor de texto, respetando las mayúsculas y las minúsculas. Puedes guardar el archivo con extensión `.Java` en la ubicación que prefieras. Recuerda que el nombre de la clase principal (en el código de ejemplo `MiModulo`) debe ser exactamente igual al del archivo con extensión `.Java`, si tienes esto en cuenta la aplicación podrá ser compilada correctamente y ejecutada.

```
/**  
 * La clase MiModulo implementa una aplicación que  
 * simplemente imprime "Módulo profesional - Programación" en pantalla.  
 */  
class MiModulo {  
  
    public static void main(String[] args) {  
        System.out.println("Módulo profesional - Programación"); // Muestra la cadena de caracteres.  
    }  
}
```

Accede a la línea de comandos y teclea, en la carpeta donde has guardado el archivo Java, el comando **para compilarlo**:

```
javac MiModulo.java
```

El compilador genera entonces un fichero de código de bytes: `MiModulo.class`. Si visualizas ahora el contenido de la carpeta verás que en ella está el archivo `.Java` y uno o varios (depende de las clases que contenga el archivo con el código fuente) archivos `.class`.

Finalmente, **para realizar la ejecución** del programa debes utilizar la siguiente sentencia:

```
java MiModulo
```

Si todo ha ido bien, verás escrito en pantalla: "Módulo profesional – Programación".

## 8.6.- Tipos de aplicaciones en Java.

La versatilidad del lenguaje de programación Java permite al programador crear distintos tipos de aplicaciones. A continuación, describiremos las características más relevantes de cada uno de ellos:



Oxygen team (GNU/GPL)

### ✓ Aplicaciones de consola:

- ◆ Son programas independientes al igual que los creados con los lenguajes tradicionales.
- ◆ Se componen como mínimo de un archivo `.class` que debe contar necesariamente con el método `main`.
- ◆ No necesitan un navegador web y se ejecutan cuando invocamos el comando Java para iniciar la Máquina Virtual de Java (JVM). De no encontrarse el método `main` la aplicación no podrá ejecutarse.
- ◆ Las aplicaciones de consola leen y escriben hacia y desde la entrada y salida estándar, sin ninguna interfaz gráfica de usuario.

### ✓ Aplicaciones gráficas:

- ◆ Aquellas que utilizan las clases con capacidades gráficas, como Swing que es la biblioteca para la interfaz gráfica de usuario avanzada de la plataforma Java SE.
- ◆ Incluyen las instrucciones `import`, que indican al compilador de Java que las clases del paquete `Javax.swing` se incluyan en la compilación.

### ✓ Applets:

- ◆ Son programas incrustados en otras aplicaciones, normalmente una página web que se muestra en un navegador. Cuando el navegador carga una web que contiene un applet, éste se descarga en el navegador web y comienza a ejecutarse. Esto nos permite crear programas que cualquier usuario puede ejecutar con tan solo cargar la página web en su navegador.
- ◆ Se pueden descargar de Internet y se observan en un navegador. Los applets se descargan junto con una página HTML desde un servidor web y se ejecutan en la máquina cliente.
- ◆ No tienen acceso a partes sensibles (por ejemplo: no pueden escribir archivos), a menos que uno mismo le dé los permisos necesarios en el sistema.
- ◆ No tienen un método principal.
- ◆ Son multiplataforma y pueden ejecutarse en cualquier navegador que soporte Java.

### ✓ Servlets:

- ◆ Son componentes de la parte del servidor de Java EE, encargados de generar respuestas a las peticiones recibidas de los clientes.
- ◆ Los servlets, al contrario de los applets, son programas que están pensados para trabajar en el lado del servidor y desarrollar aplicaciones Web que interactúen con los clientes.

### ✓ Midlets:

- ◆ Son aplicaciones creadas en Java para su ejecución en sistemas de propósito simple o dispositivos móviles. Los juegos Java creados para teléfonos móviles son midlets.
- ◆ Son programas creados para dispositivos embebidos (se dedican a una sola actividad), más específicamente para la máquina virtual Java MicroEdition (Java ME).
- ◆ Generalmente son juegos y aplicaciones que se ejecutan en teléfonos móviles.

## Autoevaluación

Un Applet es totalmente seguro ya que no puede acceder, en ningún caso, a zonas sensibles del sistema. Es decir, no podría borrar o modificar nuestros archivos.

Verdadero  Falso

**Falso**

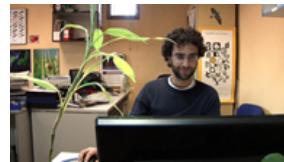
Los Applets podrían acceder a zonas sensibles de nuestro sistema si les diéramos permisos para hacerlo. Pero si no está firmado como confiable, tiene un acceso limitado al sistema del usuario.

## 9.- Entornos Integrados de Desarrollo (IDE).

### Caso práctico

**Ada, Juan y María** están navegando por Internet buscando información sobre herramientas que les faciliten trabajar en Java. **Ada** aconseja utilizar alguno de los entornos de desarrollo integrado existentes, ya que las posibilidades y rapidez que ofrecen, aumentarían la calidad y reducirían el tiempo requerido para desarrollar sus proyectos.

**Juan**, que está chateando con un miembro de un foro de programadores al que pertenece, corrobora lo que **Ada** recomienda.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

En los comienzos de Java la utilización de la línea de comandos era algo habitual. El programador escribía el código fuente empleando un editor de texto básico, seguidamente, pasaba a utilizar un compilador y con él obtenía el código compilado. En un paso posterior, necesitaba emplear una tercera herramienta para el ensamblado del programa. Por último, podía probar a través de la línea de comandos el archivo ejecutable. El problema surgía cuando se producía algún error, lo que provocaba tener que volver a iniciar el proceso completo.

Estas circunstancias hacían que el desarrollo de software no estuviera optimizado. Con el paso del tiempo, se fueron desarrollando aplicaciones que incluían las herramientas necesarias para realizar todo el proceso de programación de forma más sencilla, fiable y rápida. Para cada lenguaje de programación existen múltiples entornos de desarrollo, cada uno con sus ventajas e inconvenientes. Dependiendo de las necesidades de la persona que va a programar, la facilidad de uso o lo agradable que le resulte trabajar con él, se elegirá entre unos u otros entornos.

Para el lenguaje de programación Java existen múltiples alternativas, siendo los principales entornos de desarrollo **NetBeans** (que cuenta con el apoyo de la empresa Sun), **Eclipse** y **JCreator**. Los dos primeros son gratuitos, con soporte de idiomas y multiplataforma (Windows, Linux, MacOS).

¿Y cuál será con el que vamos a trabajar? El entorno que hemos seleccionado llevar a cabo nuestros desarrollos de software en este módulo profesional será **NetBeans**, al haber sido construido por la misma compañía que creó Java, ser de código abierto y ofrecer capacidades profesionales. Aunque, no te preocupes, también haremos un recorrido por otros entornos destacables.

## 9.1.- ¿Qué son?

Son aplicaciones que ofrecen la posibilidad de llevar a cabo el proceso completo de desarrollo de software a través de un único programa. Podremos realizar las labores de edición, compilación, depuración, detección de errores, corrección y ejecución de programas escritos en Java o en otros lenguajes de programación, bajo un entorno gráfico (no mediante línea de comandos). Junto a las capacidades descritas, cada entorno añade otras que ayudan a realizar el proceso de programación, como por ejemplo: código fuente coloreado, plantillas para diferentes tipos de aplicaciones, creación de proyectos, etc.

Hay que tener en cuenta que un entorno de desarrollo no es más que una fachada para el proceso de compilación y ejecución de un programa. ¿Qué quiere decir eso? Pues que si tenemos instalado un IDE y no tenemos instalado el compilador, no tenemos nada.



Sasa Stefanovic (GNU/GPL)

### Para saber más

Si deseas conocer algo más sobre lo que son los Entornos Integrados de Desarrollo (IDE) accede a las definiciones que te proponemos a continuación:

[Definición de Entorno Integrado de Desarrollo en Wikipedia](#)

Recuerda que cursarás un Módulo Profesional denominada "Entornos de Desarrollo", donde aprenderás a utilizar toda la funcionalidad de los IDEs.

## 9.2.- IDE's actuales.

Existen en el mercado multitud de entornos de desarrollo para el lenguaje Java, los hay de libre distribución, de pago, para principiantes, para profesionales, que consumen más recursos, que son más ligeros, más amigables, más complejos que otros, etc.

Entre los que son gratuitos o de libre distribución tenemos:

- ✓ NetBeans
- ✓ Eclipse
- ✓ BlueJ
- ✓ Jgrasp
- ✓ Jcreator LE

Entre los que son propietarios o de pago tenemos:

- ✓ IntelliJ IDEA
- ✓ Jbuilder
- ✓ Jcreator
- ✓ JDeveloper

### Debes conocer

Cada uno de los entornos nombrados más arriba posee características que los hacen diferentes unos de otros, pero para tener una idea general de la versatilidad y potencia de cada uno de ellos, accede a la siguiente tabla comparativa:

[Comparativa entornos para Java](#)

Pero, ¿cuál o cuáles son los más utilizados por la comunidad de programadores Java? El puesto de honor se lo disputan entre **Eclipse**, **IntelliJ IDEA** y **NetBeans**. En los siguientes epígrafes haremos una descripción de NetBeans y Eclipse, para posteriormente desarrollar los puntos claves del entorno NetBeans.

### Para saber más

Si quieres conocer un poco más sobre los IDEs mas utilizados en el desarrollo de Java accede al siguiente enlace:

[IDEs mas utilizados para Java](#)

### Autoevaluación

¿Cuál de los siguientes entornos sólo está soportado en la plataforma Windows?

- Eclipse.
- IntelliJ IDEA.
- Jcreator.

El nivel de expansión de Eclipse se ha visto potenciado por estar soportado en la mayoría de plataformas. Aunque no tiene un entorno visual de desarrollo (GUI Builder).

Este entorno está soportado en todas las plataformas e incorpora un entorno visual de desarrollo.

Este entorno, además de ser de pago, sólo es soportado en la plataforma Windows. No incorpora un entorno visual de desarrollo.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

## 9.3.- El entorno NetBeans.

Como se ha indicado anteriormente, el entorno de desarrollo que vamos a utilizar a lo largo de los contenidos del módulo profesional será **NetBeans**. Por lo que vamos primero a analizar sus características y destacar las ventajas que puede aportar su utilización.

Se trata de un entorno de desarrollo orientado principalmente al lenguaje Java, aunque puede servir para otros lenguajes de programación. Es un producto libre y gratuito sin restricciones de uso. Es un proyecto de código abierto de gran éxito, con una comunidad de usuarios numerosa, en continuo crecimiento y apoyado por varias empresas.

El origen de este entorno hay que buscarlo en un proyecto realizado por estudiantes de la República Checa. Fue el primer IDE creado en lenguaje Java. Un tiempo más tarde, se formó una compañía que sería comprada en 1999 por Sun Microsystems (quien había creado el lenguaje Java). Poco después, Sun decidió que el producto sería libre y de código abierto y nació Netbeans como IDE de código abierto para crear aplicaciones Java.

NetBeans lleva tiempo pugnando con Eclipse por convertirse en la plataforma más importante para crear aplicaciones en Java. Hoy en día es un producto en el que participan decenas de empresas con Sun a la cabeza. Sigue siendo software libre y ofrece las siguientes posibilidades:

- ✓ Escribir código en C, C++, Ruby, Groovy, Javascript, CSS y PHP además de Java.
- ✓ Permitir crear aplicaciones J2EE gracias a que incorpora servidores de aplicaciones Java (actualmente Glassfish y Tomcat).
- ✓ Crear aplicaciones Swing de forma sencilla, al estilo del Visual Studio de Microsoft.
- ✓ Crear aplicaciones JME para dispositivos móviles.

La última versión lanzada en junio de 2020 es la **Apache NetBeans 12.0 LTS**. Se trata de una versión LTS (*Long Time Support*), es decir, con soporte de al menos tres años aunque salgan nuevas versiones.



[Apache Netbeans](#) (Apache License)

La plataforma NetBeans permite que las aplicaciones sean desarrolladas a partir de un conjunto de componentes de software llamados módulos. Un módulo es un archivo Java que contiene clases de Java escritas para interactuar con las APIs de NetBeans y un archivo especial (manifest file) que lo identifica como módulo.

Las aplicaciones construidas a partir de módulos pueden ser extendidas agregándole nuevos módulos. Debido a que los módulos pueden ser desarrollados independientemente, las aplicaciones basadas en esta plataforma pueden ser extendidas fácilmente por cualquiera que desarrolle también software.

Cada módulo provee una función bien definida, tales como el soporte de Java, edición, o soporte para el sistema de control de versiones. NetBeans contiene todos los módulos necesarios para el desarrollo de aplicaciones Java en una sola descarga, permitiendo a la persona que va a realizar el programa comenzar a trabajar inmediatamente.

### Para saber más

Encuentra toda la información sobre el entorno de desarrollo Apache Netbeans 12 en el siguiente enlace (en inglés).



## 9.4.- Instalación y configuración.

En la siguiente presentación podrás observar el proceso de instalación de Apache Netbeans 12. Debes tener en cuenta que Netbeans está escrito en Java por lo tanto para su instalación tenemos que tener instalada el JRE. Además, como nuestra intención en programar en lenguaje Java, también necesitamos el JDK. Ambas herramientas ya las tenemos instaladas.

◀ 1 2 3 4 5 6 ▶

### Instalación de Apache Netbeans 12

Accedemos a la web de descarga de Apache Netbeans 12. Web de [Descarga Apache Netbeans 12](#)



#### Apache NetBeans Releases

Apache NetBeans is released four times a year. For details, see [full release schedule](#).

Our annual May/June release is a long-term support (LTS) release that benefits from our [NetCAT community testing process](#), and remains available and supported for a year. Our other quarterly releases provide early access to new features, which are tested and consolidated in the subsequent LTS release.

#### Apache NetBeans 12 LTS (NB 12.0)

Latest LTS version of the IDE, released on June 4, 2020.

[Features](#) [Download](#)

### Instalación de Apache Netbeans 12 II

Seleccionamos el fichero a descargar dependiendo del sistema operativo en el cual vayamos a instalarlo.

#### Downloading Apache NetBeans 12.0

Apache NetBeans 12.0 was released on June 4, 2020. See [Apache NetBeans 12.0 Features](#) for a full list of features.

Apache NetBeans 12.0 is available for download from your closest Apache mirror.

- Binaries: [netbeans-12.0-bin.zip](#) (SHA-512, PGP ASC)
- Source: [netbeans-12.0-source.zip](#) (SHA-512, PGP ASC)
- Installers:
  - [Apache-NetBeans-12.0-bin-windows-x64.exe](#) (SHA-512, PGP ASC)
  - [Apache-NetBeans-12.0-bin-linux-x64.sh](#) (SHA-512, PGP ASC)
  - [Apache-NetBeans-12.0-bin-macosx.dmg](#) (SHA-512, PGP ASC)
- Javadoc for this release is available at <https://bits.netbeans.org/12.0/javadoc>

Deployment platforms

Community approval

Known problems

Earlier releases

Officially, it is important that you [verify the integrity](#) of the downloaded files using the PGP signatures (.asc file) or a hash (.sha512 files). The PGP signatures should be matched against the [KEYS](#) file which contains the PGP keys used to sign this release.

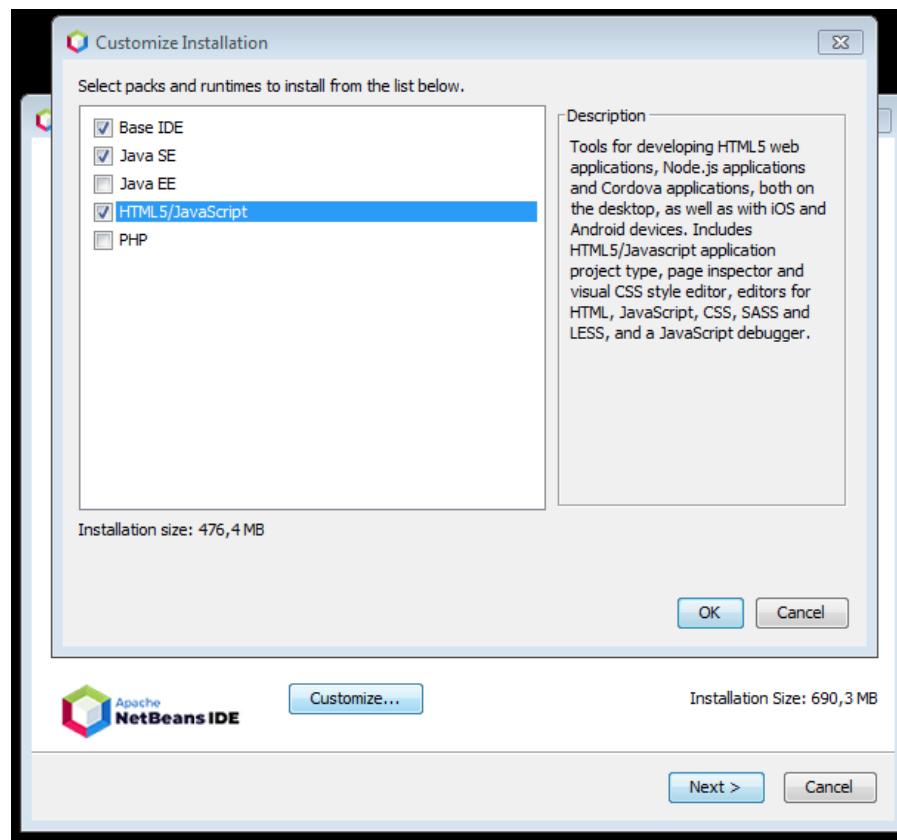
Apache NetBeans can also be installed as a self-contained [snap package](#) on Linux.

Una vez descargado (pesa mas de 300MB), iniciamos la instalación haciendo click sobre el fichero descargado. El asistente nos guiará por el sencillo proceso de instalación. Netbeans es una aplicación pesada, que consume bastantes recursos de la máquina, sobre todo memoria.

### Instalación de Apache Netbeans 12 III

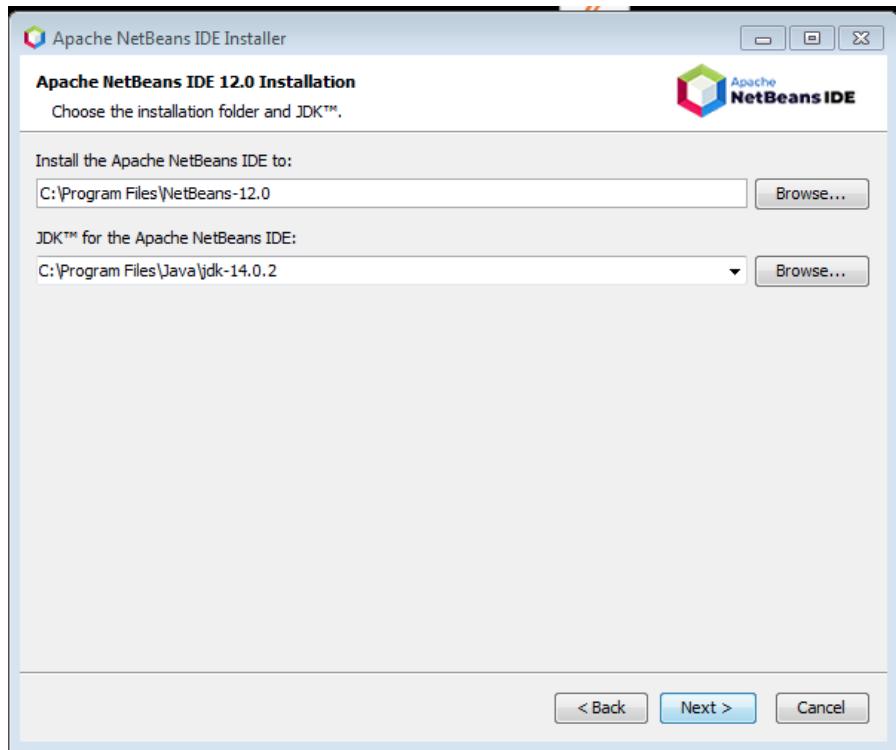
En uno de los primeros pasos de la instalación, podemos observar un botón denominado **Customize**. Pulsando sobre dicho botón podremos seleccionar a qué lenguajes queremos dar soporte en Netbeans 12. Para que

consuma menos memoria, tan solo seleccionaremos **Base IDE, Java SE y HTML5/Javascript** (nos obliga a seleccionarlo para instalar Java SE).



## Instalación de Apache Netbeans 12 IV

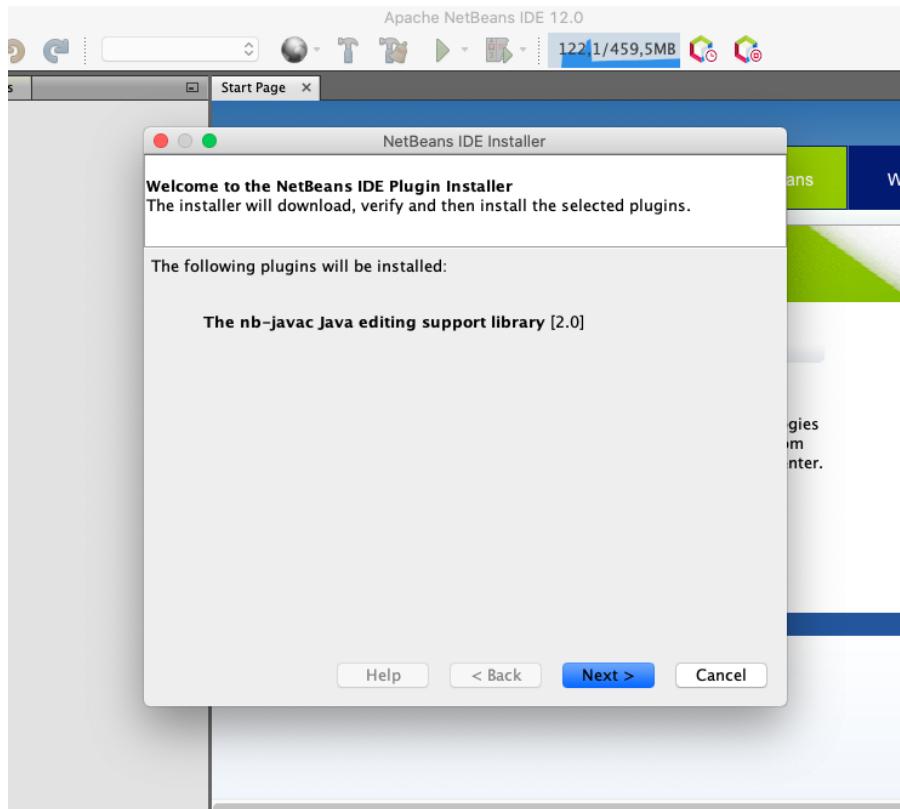
Observa como en Netbeans 12, durante el proceso de instalación, detecta el jdk que tenemos ya instalado. Si no fuera así, tendríamos que instalarlo previo a la instalación de Netbeans. Por otro lado, si la ruta no es correcta o quisierámos utilizar otro JDK (podríamos tener varios instalados), podríamos indicarlo en este paso.



¡Tras unos minutos, Apache Netbeans 12 está instalado en nuestro sistema. Ya podemos lanzarlo!

## Instalación de Apache Netbeans 12 V

Es posible que al abrir Netbeans si inicie el instalador de plugins y nos solicite la instalación del plugin **nb-javac Java editing library**. Se trata de un plugin que utiliza Netbeans para mejorar el editor de código. Debemos instalarlo. Para ello, tan solo seguimos los pasos indicados por el instalador: la descarga e instalación será automática.



## Instalación de Apache Netbeans 12 VI

Todas las imágenes utilizadas son propiedad del Ministerio de Educación y FP bajo licencia CC-BY-NC y se corresponden con capturas de pantalla.

### Para saber más

¿Utilizas como sistema operativo alguna distribución de Linux?

En el siguiente enlace tienes un manual para la instalación de Netbeans 12 en Ubuntu, Debian o Linux Mint.

[Instalación de Netbeans 12 en Linux \(en inglés\)](#)

## 9.5.- Aspecto del entorno y gestión de proyectos.

La pantalla inicial de nuestro entorno de desarrollo ofrece accesos directos a las operaciones más usuales: aprendizaje inicial, tutoriales, ejemplos, demos, los últimos programas realizados y las novedades de la versión.

Para comenzar a conocer el entorno de Apache Netbeans nada mejor que comenzar a utilizarlo. Crearemos un proyecto que mostrará por pantalla el mensaje "**Hola alumnos de Programación**".

### Debes conocer

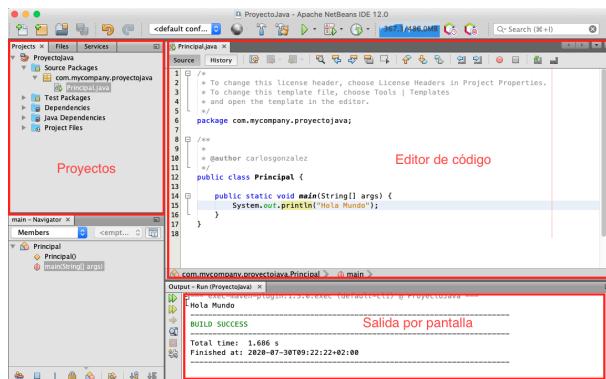
Observa en las siguientes diapositivas el proceso de creación, codificación, compilación y ejecución de nuestro primer proyecto Java.

1 2 3 4 5 6 ▶

### Mi primer proyecto en Apache Netbeans 12

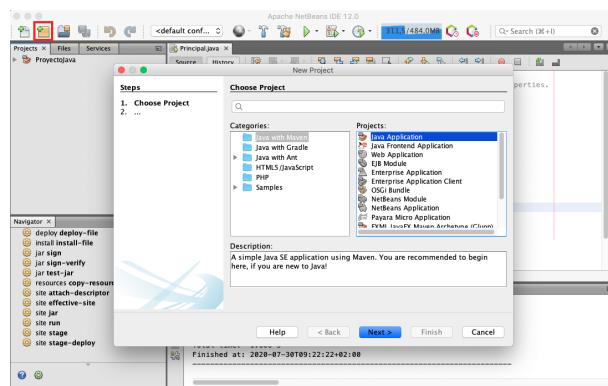
Al iniciar Netbeans, este es el entorno de trabajo en el que nos encontramos:

- **Panel Proyectos:** Desde este panel gestionaremos nuestros proyectos. En Netbeans, como en otros IDEs, toda aplicación Java está empaquetada en un proyecto. Solo podremos lanzar a ejecución una aplicación contenida en un proyecto.
- **Panel Editor de Código:** Editor de código Java.
- **Panel Output:** Este panel muestra la salida de las aplicaciones por pantalla. Nos facilita la ejecución de las aplicaciones sin necesidad de hacerlo desde una terminal.



### Mi primer proyecto en Apache Netbeans 12 II

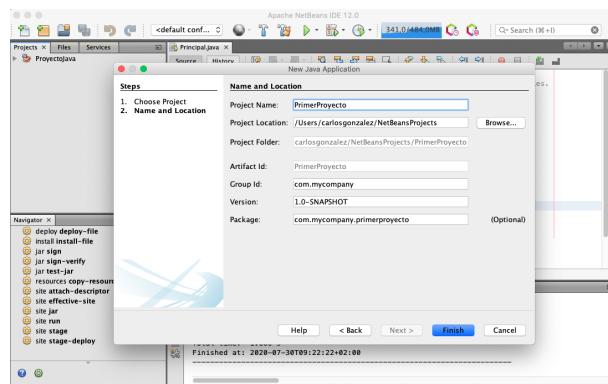
Vamos a crear un nuevo proyecto. Para ello utilizamos el icono apropiado de la barra de herramientas o pulsando con el botón derecho en el panel de proyectos y seleccionando "**New Project**". Obsérvalo:



Debemos seleccionar **Java application**, que nos permite crear una aplicación estándar Java. Como te habrás dado cuenta, existen asistentes para generar multitud de tipo de aplicaciones.

## Mi primer proyecto en Apache Netbeans 12 III

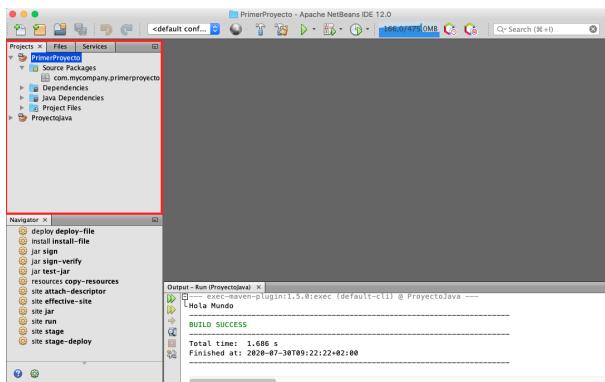
El siguiente paso es darle nombre a nuestro proyecto. Recuerda que hay que utilizar nombres lo mas descriptivos posibles, pues en algún momento tendremos muchos proyectos en nuestro panel de proyectos.



Por ahora, todas las opciones las dejamos por defecto. Pulsamos en **Finish**.

## Mi primer proyecto en Apache Netbeans 12 II

Observa en la siguiente imagen la estructura del proyecto que ha creado Netbeans.



En la carpeta **Source Packages**, Netbeans almacena los ficheros con el código fuente de nuestra aplicación, es decir, los ficheros .java. Nuestro proyecto aún no contiene ficheros fuente.

## Mi primer proyecto en Apache Netbeans 12 II

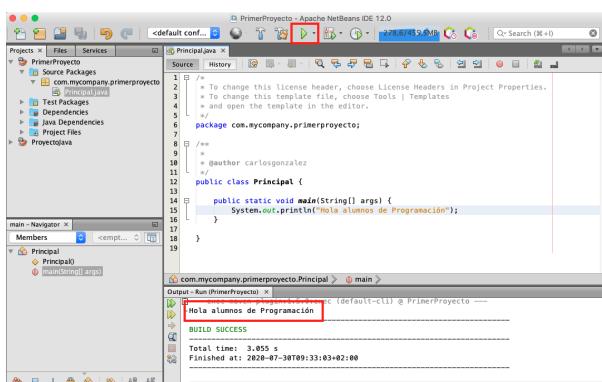
En la siguiente imagen puedes observar el proyecto con un fichero fuente denominado **Principal.java** que contiene el código necesario para mostrar por pantalla el mensaje "*Hola alumnos de programación*". Para crearlo tan solo tendrás que clickear con el botón derecho sobre el paquete existente y en el menú contextual seleccionar **New - Java Class**. En la ventana que aparece asígnale un nombre y pulsa **Finish**. Añade a tu clase el código necesario según la imagen.

El siguiente paso sería compilar y ejecutar el proyecto. En Netbeans, el proceso de compilación se realiza automáticamente al escribir el código y guardar, es decir, no tenemos que lanzarlo explícitamente (aunque también es posible hacerlo). Además, el editor marca las líneas de código que escribimos que contienen errores. Lo veremos en la siguiente unidad aunque debes asegurarte que en tu clase no existan líneas marcadas de rojo como en la siguiente imagen:



El código mostrado tiene un error sintáctico en la línea 15, pues la sentencia **println** no está bien escrita (contiene un espacio en blanco).

Observa en la siguiente imagen cómo ejecutar el proyecto y ver la salida por pantalla.



El botón **Run Project** de la barra de herramientas (también seleccionable desde el menú o la tecla F6) nos permite ejecutar nuestro proyecto. Si el código es correcto, Netbeans:

- Analiza el código para comprobar que es correcto.

- Compila el código fuente y genera los ficheros **.class** con los bytecodes Java.
- Ejecuta la aplicación y muestra la salida en el panel **Output**, sin necesidad de seleccionar el fichero **.class**.

**¡Como puedes comprobar, los IDEs son mas útiles que un simple editor de texto pues nos facilitan todas las funciones que tenemos que realizar como programadores!**

## Mi primer proyecto en Apache Netbeans 12 II

Todas las imágenes utilizadas son propiedad del Ministerios de Educación y FP (CC BY-NC) y contienen captura de pantalla de la aplicación Netbeans, propiedad de Oracle.

[Resumen textual alternativo](#)

Una de las ventajas que ofrece este entorno es poder examinar nuestros proyectos a través de la vista **Archivos**. Esta vista nos enseña la realidad de los archivos del proyecto, la carpeta **build** contiene los archivos compilados (**.class**), la carpeta **src** el código fuente y el resto, son archivos creados por Netbeans para comprobar la configuración del proyecto o los archivos necesarios para la correcta interpretación del código en otros sistemas (en cualquier caso no hay que borrarlos). Para activar esta vista, selecciona en el menú principal **Windows - Files**.

## Autoevaluación

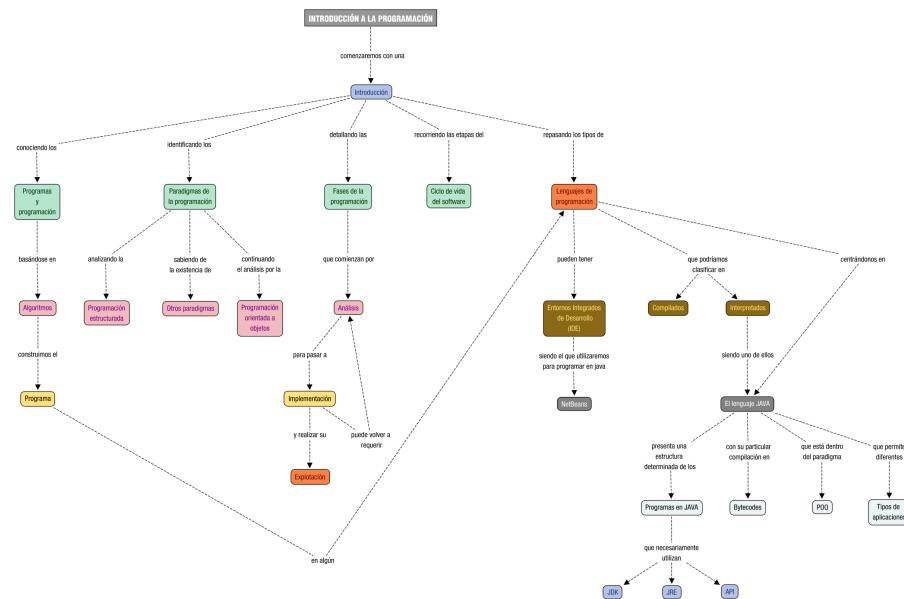
**Rellena los huecos con los conceptos adecuados:**

En NetBeans, los archivos **.class** de un proyecto están alojados en la carpeta  y los **.Java** en la carpeta .

Los archivos correspondientes a las clases se alojan en la carpeta **build** y los archivos con el código fuente se alojan en la carpeta **src**.

# 10.- Conclusiones

A lo largo de esta primera unidad hemos realizado una introducción al mundo de la programación de aplicaciones. Hemos definido algoritmo y hemos hablado sobre las fases que se llevan a cabo para el desarrollo de una aplicación software. Además, hemos introducido el concepto de lenguaje de programación y hemos enumerado los diferentes tipos existentes. Por último, hemos realizado una introducción al lenguaje Java, uno de los lenguajes orientado a objetos más utilizado en la actualidad. Además, hemos descrito las herramientas necesarias para programar en Java, desde herramientas básicas de consola hasta entornos de desarrollo de aplicaciones. El mapa conceptual resume los conceptos que deben quedar claros antes de comenzar con la segunda unidad.



En la segunda unidad nos centraremos en conocer el lenguaje Java más en profundidad. El objetivo será conocer los tipos de datos primitivos utilizados por él mismo, variables y constantes así como los operadores utilizados para la manipulación de los datos.

# Creación de mi primer programa.

## Caso práctico



Ministerio de Educación y FP [\(CC BY-NC\)](#)

Todos los lenguajes de programación están constituidos por elementos concretos que permiten realizar las operaciones básicas del lenguaje, como tipos de datos, operadores e instrucciones. Estos conceptos deben ser dominados por la persona que desee incorporarse, con ciertas garantías, a un equipo de programación. Debemos tener en cuenta que los programas trabajan con datos, los cuales almacenan en memoria y son posteriormente usados para la toma de decisiones en el programa.

En esta unidad se introducen los distintos tipos de datos que se pueden emplear en Java. En concreto, se verán los tipos primitivos en Java, así como las variables y las constantes. En posteriores unidades veremos otros elementos básicos del lenguaje, incluyendo estructuras de datos más sofisticadas.

**María** y **Juan** han formado equipo para desarrollar una aplicación informática para una clínica veterinaria. **Ada** ha convocado una reunión con el cliente para concretar los requerimientos de la aplicación, pero lo que está claro es que debe ser multiplataforma. El lenguaje escogido ha sido Java.

**María** tiene conocimientos de redes y de páginas web pero está floja en programación. **Juan** ha aprendido Java en su ciclo de DAI hace 4 años.

—Lo que hace falta es entender bien los conceptos de programación orientada a objetos —le comenta **Juan** a **María**. —Todo lo concerniente al mundo real puede ser modelado a través de objetos. Un objeto contiene datos y sobre él se hacen operaciones y gira toda la aplicación.

**María** está entusiasmada con el proyecto, cree que es una buena oportunidad para aprender un lenguaje de la mano de **Juan** que se le da bastante bien todo el mundo de la programación.



[Ministerio de Educación y Formación Profesional. \(Dominio público\)](#)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Introducción.

Cada vez que usamos un ordenador, estamos ejecutando varias aplicaciones que nos permiten realizar ciertas tareas. Por ejemplo, en nuestro día a día, usamos el correo electrónico para enviar y recibir correos, o el navegador para consultar páginas en Internet; ambas actividades son ejemplos de programas que se ejecutan en un ordenador.

Los programas de ordenador deben resolver un problema, para lo cual debemos utilizar de forma inteligente y lógica todos los elementos que nos ofrece el lenguaje. Por eso es importante elegir un lenguaje de programación con el que nos sintamos cómodos porque lo dominemos suficientemente y, por supuesto, porque sepamos que no va a ofrecer limitaciones a la hora de desarrollar aplicaciones para diferentes plataformas.

El lenguaje que vamos a utilizar en este módulo es Java. Es un lenguaje multiplataforma, robusto y fiable. Un lenguaje que reduce la complejidad y se considera dentro de los lenguajes modernos orientados a objetos. Esta unidad nos vamos a adentrar en su sintaxis, vamos a conocer los tipos de datos con los que trabaja, las operaciones que tienen definidas cada uno de ellos, utilizando ejemplos sencillos que nos muestren la utilidad de todo lo aprendido.

Para ello, vamos a tratar sobre cómo se almacenan y recuperan los datos de variables y cadenas en Java, y cómo se gestionan estos datos desde el punto de vista de la utilización de operadores. Trabajar con datos es fundamental en cualquier programa. Aunque ya hayas programado en este lenguaje, échale un vistazo al contenido de esta unidad, porque podrás repasar muchos conceptos.



Federico Romero (CC BY-NC-ND)

## Para saber más

Ahora que vamos a empezar con la sintaxis de Java, quizás te interese tener a mano la documentación que ofrece la página web de Oracle sobre [Java SE](#). La plataforma Java SE está formada principalmente por dos productos: el [JDK](#), que contiene los compiladores y depuradores necesarios para programar, y el [JRE](#), que proporciona las librerías o bibliotecas y la [JVM](#), entre otra serie de componentes.

Puedes consultar información de la versión 14 de Java SE en el siguiente enlace. Encontrarás toda la documentación sobre esta tecnología (en inglés):

[Documentación de Oracle sobre Java SE](#)

Dentro de la documentación de Oracle sobre Java SE se encuentra el libro [The Java Language Specification](#). Este libro está escrito por los inventores del lenguaje, y constituye una referencia técnica casi obligada sobre el mismo. Como mucha de la documentación oficial de Java, se encuentra en inglés. El enlace directo es el siguiente:

[The Java Language Specification](#)

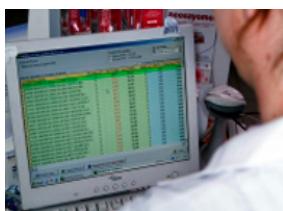
## Recomendación

Acostúmbrate a leer y consultar la documentación sobre la versión de Java que estés utilizando en tus programas. Eso te ayudará a saber todas las posibilidades que tiene el lenguaje, y si en un momento dado estás utilizando bien una determinada característica.

**Los manuales de referencia del lenguaje no se aprenden, se consultan: eso nos ayudará a ir conociendo toda la potencia de un lenguaje de programación.**

## 2.- Las variables e identificadores.

### Caso práctico



Ministerio de Educación. Aplicación Informática  
(CC BY-NC)

**María** y **Juan** han comprobado que una aplicación informática es un trabajo de equipo que debe estar perfectamente coordinado. El primer paso es la definición de los datos y las variables que se van a utilizar. Las decisiones que se tomen pueden afectar a todo el proyecto, en lo referente al rendimiento de la aplicación y ahorro de espacio en los sistemas de almacenamiento.

Después de la última reunión del equipo de proyecto ha quedado claro cuáles son las especificaciones de la aplicación a desarrollar. **Juan** no quiere perder el tiempo y ha comenzado a preparar los datos que va a usar el programa. Le ha pedido a **María** que vean juntos qué variables y tipos de datos se van a utilizar, **Juan** piensa que le vendrá bien como primera tarea para familiarizarse con el entorno de programación y con el lenguaje en sí.

Un programa maneja datos para hacer cálculos, presentarlos en informes por pantalla o impresora, solicitarlos al usuario, guardarlos en disco, etc. Para poder manejar esos datos, el programa los guarda en variables.

Una ..... variable es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa.

Las variables vienen determinadas por:

- ✓ Un nombre, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.
- ✓ Un ..... tipo de dato, que especifica qué clase de información guarda la variable en esa zona de memoria
- ✓ Un rango de valores que puede admitir dicha variable, que vendrá determinado por el tipo de dato.

Al nombre que le damos a la variable se le llama **identificador**. Los identificadores permiten nombrar los elementos que se están manejando en un programa. Vamos a ver con más detalle ciertos aspectos sobre los identificadores que debemos tener en cuenta.

Las variables se declaran de la siguiente forma:

```
tipoVariable identificadorVariable;
```

Por ejemplo:

```
int aux;
```

Que se interpreta como: se ha definido una variable de tipo entero, que ocupará en memoria RAM 4 Bytes (para permitir almacenar números enteros grandes) cuyo identificador es **aux**. El tamaño que ocupa en memoria una determinada variable dependerá de su tipo y estará predefinido en el lenguaje.

### Citas para pensar

Las grandes ideas requieren un gran lenguaje. Aristófanes.

## Para saber más

Bruce Eckel es el autor de los libros sobre Java y C++, dirigidos a programadores que desean aprender sobre estos lenguajes y sobre la programación orientada a objetos. Este escritor ha tenido la buena costumbre de editar sus libros para que puedan descargarse gratis. Así, podemos acceder de forma totalmente gratuita a la cuarta edición de su libro **Thinking in Java** en el siguiente enlace (en inglés) y una versión traducida al español. No te asustes, se trata de manuales de referencia del lenguaje, que se utilizan para consulta en caso de duda:

[Libro "Thinking in Java"](#)

[Libro "Piensa en Java", cuarta edición, versión traducida en pdf](#)

A partir de ahora es conveniente que utilices algún manual de apoyo para iniciarte a la programación en Java. Te proponemos el de la serie de Libros **Aprenda Informática como si estuviera en primero**, de la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra):

[Manual de apoyo sobre Java](#)

## 2.1.- Identificadores.



svensson, Unicode (CC BY-NC)

Un **identificador** en Java es una secuencia ilimitada sin espacios de letras y dígitos Unicode, de forma que **el primer símbolo de la secuencia debe ser una letra, un símbolo de subrayado (\_) o el símbolo dólar (\$)**. Por ejemplo, son válidos los siguientes identificadores:

x5      ατη      NUM\_MAX      numCuenta

En la definición anterior decimos que un identificador es una secuencia ilimitada de caracteres Unicode. Pero... ¿qué es Unicode? Unicode es un código de caracteres o sistema de codificación, un alfabeto que recoge los caracteres de prácticamente todos los idiomas importantes del mundo.

Las líneas de código en los programas se escriben usando ese conjunto de caracteres Unicode.

Esto quiere decir que en Java se pueden utilizar varios alfabetos como el Griego, Árabe o Japonés. De esta forma, los programas están más adaptados a los lenguajes e idiomas locales, por lo que son más significativos y fáciles de entender tanto para los programadores que escriben el código, como para los que posteriormente lo tienen que interpretar, para introducir alguna nueva funcionalidad o modificación en la aplicación.

El estándar Unicode originalmente utilizaba 16 bits, pudiendo representar hasta 65.536 caracteres distintos, que es el resultado de elevar dos a la potencia dieciséis. Actualmente Unicode puede utilizar más o menos bits, dependiendo del formato que se utilice: **UTF-8, UTF-16 ó UTF-32**. A cada carácter le corresponde únicamente un número entero perteneciente al intervalo de 0 a 2 elevado a n, siendo n el número de bits utilizados para representar los caracteres. Por ejemplo, la letra ñ es el entero 164. Además, el código Unicode es “compatible” con el código ASCII, ya que para los caracteres del código ASCII, Unicode asigna como código los mismos 8 bits, a los que les añade a la izquierda otros 8 bits todos a cero. La conversión de un carácter ASCII a Unicode es inmediata.



Mª Flor Moncada Añón (Elaboración Propia) (CC BY-NC)

### Recomendación

Una buena práctica de programación es seleccionar nombres adecuados para las variables, que sean lo más descriptivos posible. Eso ayuda a que el programa se autodocumente y evita un número excesivo de comentarios para aclarar el código. **¡Es muy conveniente hacer el esfuerzo de pensar en un identificador apropiado para las variables!**

### Para saber más

Enlace para acceder a la documentación sobre las distintas versiones de Unicode en la página web oficial del estándar:

[Documentación sobre Unicode](#)

## 2.2.- Convenios y reglas para nombrar variables.

A la hora de nombrar un identificador existen una serie de normas de estilo de uso generalizado que, no siendo obligatorias, se usan en la mayor parte del código Java. Estas reglas para la nomenclatura de variables son las siguientes:

- ✓ **Java distingue las mayúsculas de las minúsculas.** Por ejemplo, Alumno y alumno son variables diferentes.
- ✓ **No se suelen utilizar identificadores que comiencen con «\$» o «\_»,** además el símbolo del dólar, por convenio, no se utiliza nunca.
- ✓ **No se puede utilizar el valor booleano (true O false) ni el valor nulo (null).**
- ✓ **Los identificadores deben ser lo más descriptivos posibles.** Es mejor usar palabras completas en vez de abreviaturas críticas. Así nuestro código será más fácil de leer y comprender. En muchos casos también hará que nuestro código se autodocumente. Por ejemplo, si tenemos que darle el nombre a una variable que almacena los datos de un cliente sería recomendable que la misma se llamara algo así como FicheroClientes O ManejadorCliente, y no algo poco descriptivo como c133.



Mº Flor Moncada Añón (Elaboración Propia) [CC BY-NC](https://creativecommons.org/licenses/by-nc/4.0/)

Además de estas restricciones, en la siguiente tabla puedes ver otras convenciones, que no siendo obligatorias, sí son recomendables a la hora de crear identificadores en Java.

### Convenciones sobre identificadores en Java

Identificador	Convención	Ejemplo
Nombre de variable.	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas.	numAlumnos, suma
Nombre constante.	En letras mayúsculas, separando las palabras con el guión bajo, por convenio el guión bajo no se utiliza en ningún otro sitio.	TAM_MAX, PI
Nombre de una clase.	Comienza por letra mayúscula.	String, MiTipo
Nombre función.	Comienza con letra minúscula.	modifica_Valor, obtiene_Valor

### Autoevaluación

Un identificador es una secuencia de uno o más símbolos Unicode que cumple las siguientes condiciones. Señala la afirmación correcta.

- Todos los identificadores han de comenzar con una letra, el carácter subrayado (\_) o el carácter dólar (\$).
- No puede incluir el carácter espacio en blanco.
- No pueden incluir el valor booleano false.
- Todas las anteriores son correctas.

Incorrecto, existen más condiciones.

Incorrecta, sigue intentándolo.

No es la única respuesta correcta.

¡Exacto! Además, se desaconseja el uso del símbolo dólar, y el guión bajo prácticamente sólo se utiliza para separar palabras en variables de tipo Constante.

## Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

## 2.3.- Palabras reservadas.

Las palabras reservadas, a veces también llamadas palabras clave o keywords , son secuencias de caracteres formadas con letras ASCII cuyo uso se reserva al lenguaje y, por tanto, **no pueden utilizarse para crear identificadores**.

Las palabras reservadas en Java son:

Palabras clave en Java

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

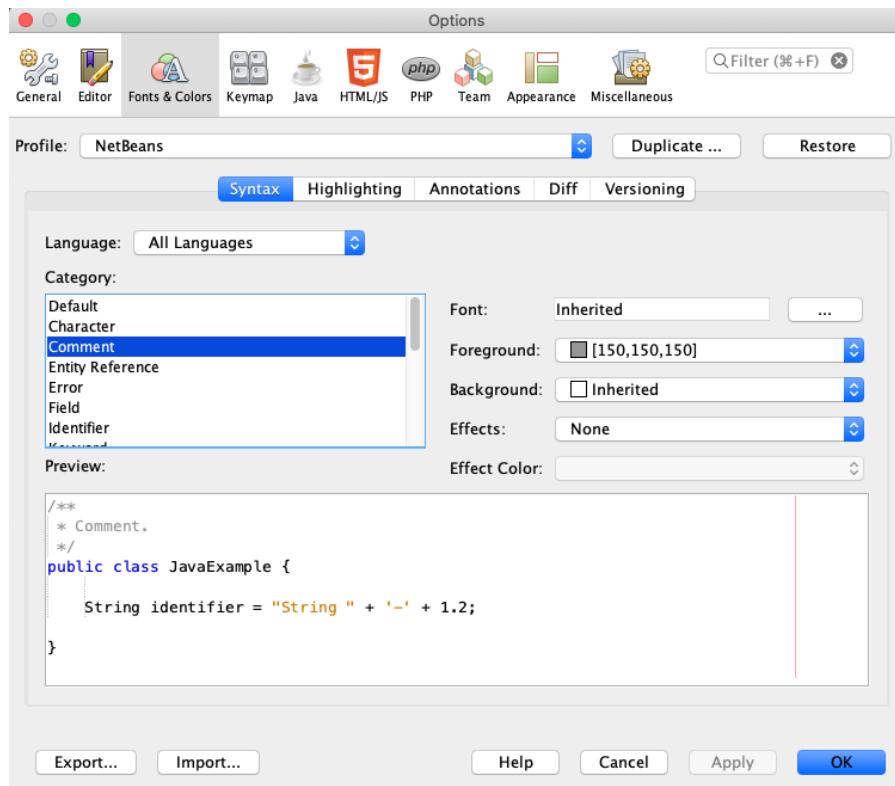


[ytueresburroyyomemonto \(CC BY-NC\)](#)

Hay palabras reservadas que no se utilizan en la actualidad, como es el caso de `const` y `goto`, que apenas se utilizan en la actual implementación del lenguaje Java. Por otro lado, puede haber otro tipo de palabras o texto en el lenguaje que aunque no sean palabras reservadas tampoco se pueden utilizar para crear identificadores. Es el caso de `true` y `false` que, aunque puedan parecer palabras reservadas, porque no se pueden utilizar para ningún otro uso en un programa, técnicamente son **literales booleanos**. Igualmente, `null` es considerado un literal, no una palabra reservada.

Normalmente, los editores y entornos integrados de desarrollo utilizan colores para diferenciar las palabras reservadas del resto del código, los comentarios, las constantes y literales, etc. De esta forma se facilita la lectura del programa y la detección de errores de sintaxis.

Aunque no es conveniente porque Netbeans utiliza colores estándar, es posible modificar los colores utilizados para la sintaxis de tus programas. Por ejemplo si quieres que los comentarios aparezcan en verde en lugar de en gris, o indicar que tienes la autoría de los mismos, en lugar de que te aparezca el nombre de usuario del sistema operativo. Para ello tendrás que acceder a la opción de menú **Tools - Options** de Netbeans.



[Resumen textual alternativo](#)

## Recomendación

Cuando tras haber consultado la documentación de Java aún no tengas seguridad de cómo funciona alguna de sus características, pruébala en tu ordenador, y analiza cada mensaje de error que te dé el compilador para corregirlo. Busca en foros de Internet errores similares para ayudarte de la experiencia de otros usuarios y usuarias. Es la mejor manera de aprender el lenguaje y coger soltura en la escritura de código.

## 2.4.- Tipos de variables.

En un programa nos podemos encontrar distintos tipos de variables. Las diferencias entre una variable y otra dependerán de varios factores, por ejemplo, el tipo de datos que representan, si su valor cambia o no durante la ejecución, o cuál es el papel que llevan a cabo. De esta forma, el lenguaje de programación Java define los siguientes tipos de variables:

1. **Variables de tipos primitivos y variables referencia**, según el tipo de información que contengan. En función de a qué grupo pertenezca la variable, tipos primitivos o tipos referenciados, podrá tomar unos valores u otros, y se podrán definir sobre ella unas operaciones u otras.

2. **Variables y constantes**, dependiendo de si su valor cambia o no durante la ejecución del programa. La definición de cada tipo sería:

✓ **Variables**. Sirven para almacenar los datos durante la ejecución del programa, pueden estar formadas por cualquier tipo de dato primitivo o referencia. Su valor puede cambiar a lo largo de la ejecución del programa. Realmente una variable representa una zona de memoria del ordenador que contiene un determinado valor (del tipo de datos de la variable) y al que se accede a través del identificador.

✓ **Constantes o variables finales**: Son aquellas variables cuyo valor no cambia a lo largo de todo el programa.

3. **Variables miembro y variables locales**, en función del lugar donde aparezcan en el programa. La definición concreta sería:

✓ **Variables miembro**: Son las variables que se crean dentro de una clase, fuera de cualquier método. Pueden ser de tipos primitivos o referencias, variables o constantes. En un lenguaje puramente orientado a objetos como es Java, todo se basa en la utilización de objetos, los cuales se crean usando clases. En la siguiente unidad veremos los distintos tipos de variables miembro que se pueden usar.

✓ **Variables locales**: Son las variables que se crean y usan dentro de un método o, en general, dentro de cualquier bloque de código. La variable deja de existir cuando la ejecución del bloque de código o el método finaliza. Al igual que las variables miembro, las variables locales también pueden ser de tipos primitivos o referencias.

### Autoevaluación

Relaciona los tipos de variables con la característica correspondiente, escribiendo el número asociado a la característica en el hueco correspondiente.

#### Ejercicio de relacionar

Las variables...	Relación	Tienen la característica de que ...
Locales.	<input type="checkbox"/>	1. Una vez inicializadas su valor nunca cambia.
Miembro.	<input type="checkbox"/>	2. Van dentro de un método.
Constantes.	<input type="checkbox"/>	3. Van dentro de una clase.

Enviar

Las variables locales, las constantes y las variables miembro tienen esas características que las definen dentro de una aplicación Java.

## 2.4.1.- Tipos de variables II.

El siguiente ejemplo muestra el código para la creación de varios tipos de variables. Como ya veremos en apartados posteriores, las variables necesitan declararse, a veces dando un valor y otras con un valor por defecto. Este programa crea una clase que contiene las siguientes variables:

- ✓ **Variable constante llamada PI:** esta variable por haberla declarado como constante no podrá cambiar su valor a lo largo de todo el programa.
- ✓ **Variable miembro llamada x:** Esta variable pertenece a la clase `ejemplovariables`. La variable x puede almacenar valores del tipo primitivo `int`. El valor de esta variable podrá ser modificado en el programa, normalmente por medio de algún otro método que se cree en la clase.
- ✓ **Variable local llamada valorantiguo:** Esta variable es local porque está creada dentro del método `obtenerX()`. Sólo se podrá acceder a ella dentro del método donde está creada, ya que fuera de él no existe.

En apartados posteriores veremos como darle más funcionalidad a nuestros programas, mostrar algún resultado por pantalla, hacer operaciones, etc. Por el momento, si ejecutas el ejemplo anterior simplemente mostrará el mensaje "GENERACIÓN CORRECTA", indicando que todo ha ido bien y el programa ha finalizado.

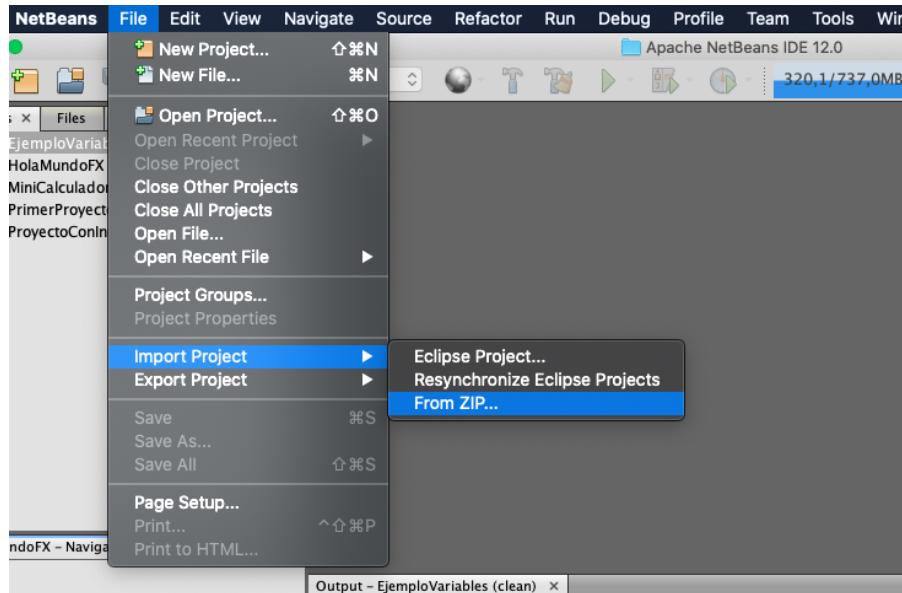
```
/* Ejemplo de tipos de variables */
public class EjemploVariables {
    public static final double PI = 3.141592653589793; // PI es una constante
    int x; // x es una variable miembro
    int obtenerX(); // obtenerX() es un método miembro
    int valorantiguo(); // valorantiguo es una variable local
    // el método main comienza la ejecución de la aplicación
    public static void main(String[] args) {
        // para irte al inicio de nuestras explicaciones
    }
}
```

Mº Flor Moncada Añón (Elaboración Propia)  
[\(CC BY-NC\)](#)

[Ejemplo de tipos de variables](#) (13.00 KB)

## Recomendación

Los programas de ejemplo suministrados están empaquetados en un fichero zip. Para abrir dicho proyecto en Netbeans, conservando su estructura y configuración debes importarlo. Netbeans permite importar proyectos desde ficheros zip, desde otros IDEs como Eclipse, etc. Observa en la imagen la opción de importar:



¡Tan solo tendrás que seleccionar el zip a importar y automáticamente tendrás el proyecto en el gestor de proyectos, listo para ser modificado o lanzado a ejecución!

### 3.- Los tipos de datos.

#### Caso práctico



Stockbyte DVD-CD (CC BY-NC)

**María** ya ha hecho sus pinitos como programadora. Ahora mismo está ayudando a Juan con las variables y le ha surgido un problema. —El lenguaje se está comportando de una forma extraña, quiero llamar a una variable final y no me deja —Le comenta a **Juan**. —Claro, eso es porque final es una palabra reservada y ya hemos visto que no la puedes utilizar para nombrar variables —le responde **Juan**. —¡Vaya! —exclama **María**, —¡es verdad!, ¿y qué otros requisitos debo tener en cuenta a la hora de declarar las variables? —Pues lo importante es saber qué tipo de información hay que guardar, para poder asignarles el tipo de dato adecuado. —Tienes un momento y te lo cuento? — le dice **Juan**.

En los ..... lenguajes fuertemente tipados, a todo dato (constante, variable o expresión) le corresponde un tipo que es conocido antes de que se ejecute el programa.

El tipo limita el valor de la variable o expresión, las operaciones que se pueden hacer sobre esos valores, y el significado de esas operaciones. Esto es así porque **un tipo de dato no es más que una especificación de los valores que son válidos para esa variable , y de las operaciones que se pueden realizar con ellos**.

Debido a que el tipo de dato de una variable se conoce durante la revisión que hace el compilador para detectar errores, o sea en tiempo de compilación, esta labor es mucho más fácil, ya que no hay que esperar a que se ejecute el programa para saber qué valores va a contener esa variable. Esto se consigue con un control muy exhaustivo de los tipos de datos que se utilizan, lo cual tiene sus ventajas e inconvenientes, por ejemplo cuando se intenta asignar un valor de un tipo, a una variable de otro tipo. Sin embargo, en Java, puede haber conversión entre ciertos tipos de datos, como veremos posteriormente.

Ahora no es el momento de entrar en detalle sobre la conversión de tipos, pero sí debemos conocer con exactitud de qué tipos de datos dispone el lenguaje Java. Ya hemos visto que las variables, según la información que contienen, se pueden dividir en variables de tipos primitivos y variables referencia. Pero ¿qué es un tipo de dato primitivo? ¿Y un tipo referencia? Esto es lo que vamos a ver a continuación. Los tipos de datos en Java se dividen principalmente en dos categorías:

- ✓ **Tipos de datos sencillos o primitivos.** Representan valores simples que vienen predefinidos en el lenguaje; contienen valores únicos, como por ejemplo un carácter o un número.
- ✓ **Tipos de datos referencia.** Se definen con un nombre o referencia (puntero) que contiene la dirección en memoria de un valor o grupo de valores. Dentro de este tipo tenemos por ejemplo los vectores o arrays, que son una serie de elementos del mismo tipo, o las clases, que son los modelos o plantillas a partir de los cuales se crean los objetos.

En el siguiente apartado vamos a ver con detalle los diferentes tipos de datos que se engloban dentro de estas dos categorías.

#### Autoevaluación

**El tipado fuerte de datos supone que:**

- A todo dato le corresponde un tipo que es conocido antes de que se ejecute el programa.
- El lenguaje hace un control muy exhaustivo de los tipos de datos.
- El compilador puede optimizar mejor el tratamiento de los tipos de datos.
- Todas las anteriores son correctas.

Incorrecto, existen más condiciones.

Incorrecta, sigue intentándolo.

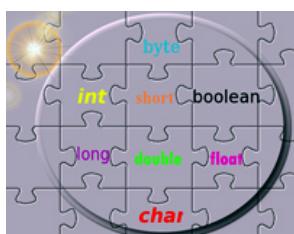
No es la única respuesta correcta.

Muy bien, en un lenguaje fuertemente tipado se cumplen todas las condiciones anteriores.

## **Solución**

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

## 3.1.- Tipos de datos primitivos I.



Mª Flor Moncada Añón (Elaboración propia) [CC BY-NC](#)

Los tipos primitivos son aquéllos datos sencillos que constituyen los tipos de información más habituales: números, caracteres y valores lógicos o booleanos.

Al contrario que en otros lenguajes de programación orientados a objetos, estas variables **en Java no son objetos**.

Una de las mayores ventajas de tratar con tipos primitivos en lugar de con objetos, es que el compilador de Java puede optimizar mejor su uso. Otra importante característica, es que cada uno de los tipos primitivos tiene **idéntico** tamaño y comportamiento en todas las versiones de Java y para cualquier tipo de ordenador. Esto quiere decir que no debemos preocuparnos de cómo se representarán los datos en distintas plataformas, y asegura la **portabilidad** de los programas, a diferencia de lo que ocurre con otros lenguajes. Por ejemplo, el tipo `int` siempre se representará con 32 bits, con signo, y en el formato de representación complemento a 2, en cualquier plataforma que soporte Java.

### Reflexiona

Java especifica el tamaño y formato de todos los tipos de datos primitivos con independencia de la plataforma o sistema operativo donde se ejecute el programa, de forma que el programador no tiene que preocuparse sobre las dependencias del sistema, y no es necesario escribir versiones distintas del programa para cada plataforma.

### Debes conocer

En el siguiente enlace se muestran los tipos de datos primitivos en Java con el rango de valores que pueden tomar, el tamaño que ocupan en memoria y sus valores por defecto.

[Tipos de Datos Primitivos en Java](#)

Hay una peculiaridad en los tipos de datos primitivos, y es que el tipo de dato `char` es considerado por el compilador como un tipo numérico, ya que los valores que guarda son el código Unicode correspondiente al carácter que representa, no el carácter en sí, por lo que puede operarse con caracteres como si se tratara de números enteros.

Por otra parte, a la hora de elegir el tipo de dato que vamos a utilizar ¿qué criterio seguiremos para elegir un tipo de dato u otro? Pues deberemos tener en cuenta cómo es la información que hay que guardar, si es de tipo texto, numérico, ... y, además, qué rango de valores puede alcanzar. En este sentido, hay veces que aunque queramos representar un número sin decimales, tendremos que utilizar datos de tipo `real`.

Por ejemplo, el tipo de dato `int` no podría representar la población mundial del planeta, ya que el valor máximo que alcanza es de 2.147.483.647, siendo éste el número máximo de combinaciones posibles con 32 bits, teniendo en cuenta que la representación de los números enteros en Java utiliza complemento a 2. Si queremos representar el valor correspondiente a la población mundial del planeta, cerca de 7 mil millones de habitantes, tendríamos que utilizar al menos un tipo de dato `long`, o si tenemos problemas de espacio un tipo `float`. Sin embargo, los tipos reales tienen otro problema: la **precisión**. Vamos a ver más sobre ellos en el siguiente apartado.

## Para saber más

Si quieres obtener información sobre cómo se lleva a cabo la representación interna de números enteros y sobre la aritmética binaria, puedes consultar el siguiente enlace:

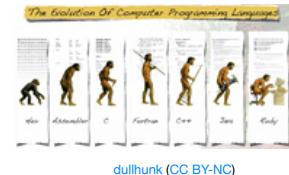
[Aritmética binaria](#)

Estos contenidos también los trabajarás en el Módulo Profesional "Sistemas Informáticos".

### 3.1.1.- Tipos de datos primitivos II.

El tipo de dato real permite representar cualquier número con decimales. Al igual que ocurre con los enteros, la mayoría de los lenguajes definen más de un tipo de dato real, en función del número de bits usado para representarlos. Cuanto mayor sea ese número:

- ✓ **Más grande podrá ser el número real representado en valor absoluto.**
- ✓ **Mayor será la precisión** de la parte decimal.



Entre cada dos números reales cualesquiera, siempre tendremos infinitos números reales, por lo que **la mayoría de ellos los representaremos de forma aproximada**. Por ejemplo, en la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333..., con la secuencia de 3 repitiéndose infinitamente. En el ordenador sólo podemos almacenar un número finito de bits, por lo que el almacenamiento de un número real será siempre una aproximación.

Los números reales se representan en coma flotante, que consiste en trasladar la coma decimal a la primera cifra significativa del valor, con objeto de poder representar el máximo de números posible.

Un número se expresa como:  $Valor = \text{mantisa} \cdot 2^{\text{exponente}}$

En concreto, sólo se almacena la **mantisa** y el exponente al que va elevada la base. Los bits empleados por la mantisa representan la **precisión** del número real, es decir, el número de cifras decimales significativas que puede tener el número real, mientras que los bits del exponente expresan la diferencia entre el mayor y el menor número representable, lo que viene a ser el **intervalo de representación**.

En Java las variables de tipo `float` se emplean para representar los números en coma flotante de simple precisión de 32 bits, de los cuales 24 son para la mantisa y 8 para el exponente. La mantisa es un valor entre -1.0 y 1.0 y el exponente representa la potencia de 2 necesaria para obtener el valor que se quiere representar. Por su parte, las variables tipo `double` representan los números en coma flotante de doble precisión de 64 bits, de los cuales 53 son para la mantisa y 11 para el exponente.

La mayoría de los programadores en Java emplean el tipo `double` cuando trabajan con datos de tipo real, es una forma de asegurarse de que los errores cometidos por las sucesivas aproximaciones sean menores. De hecho, Java considera los valores en coma flotante como de tipo `double` por defecto.

Con el objetivo de conseguir la máxima portabilidad de los programas, Java utiliza el estándar internacional ..... IEEE 754 para la representación interna de los números en coma flotante, que es una forma de asegurarse de que el resultado de los cálculos sea el mismo para diferentes plataformas.

#### Para saber más

En la siguiente página puedes encontrar documentación sobre el estándar IEEE 754. Contiene enlaces a herramientas para trabajar con el estándar en la sección "Enlaces Externos"

[Notación IEEE 754](#)

#### Autoevaluación

**Relaciona los tipos primitivos con los bits y rango de valores correspondientes, escribiendo el número asociado en el hueco correspondiente.**

##### Ejercicio de relacionar

Tipo	Relación	Característica
short	<input type="radio"/>	Coma flotante de 64 bits, usando la representación IEE754-2008

<b>Tipo</b>	<b>Relación</b>	<b>Característica</b>
<b>byte</b>	<input type="checkbox"/>	Entero de 32 bits, rango de valores de -2.147.483.648 ( $-2^{31}$ ) a 2.147.483.647 ( $+2^{31}-1$ )
<b>double</b>	<input type="checkbox"/>	Entero de 16 bits, rango de valores de -32.768 ( $-2^{15}$ ) a +32.767 ( $+2^{15}-1$ )
<b>long</b>	<input type="checkbox"/>	Coma flotante de 32 bits, usando la representación IEEE 745-2008
<b>int</b>	<input type="checkbox"/>	Entero de 8 bits, rango de valores de -128 ( $-2^7$ ) a +127 ( $+2^7-1$ )
<b>float</b>	<input type="checkbox"/>	Entero de 64 bits, rango de valores de -9.223.372.036.854.775.808 (-2 <sup>63</sup> ) a 9.223.372.036.854.775.807 (+2 <sup>63</sup> -1)

Enviar

Además de los anteriores, también son tipos primitivos el tipo de dato boolean, con valores true y false, y el tipo de datos char, que almacena el código Unicode de un carácter.

## 3.2.- Declaración e inicialización.

Llegados a este punto cabe preguntarnos ¿Cómo se crean las variables en un programa? ¿Qué debo hacer antes de usar una variable en mi programa? Pues bien, como podrás imaginar, debemos crear las variables antes de poder utilizarlas en nuestros programas, indicando qué identificador (nombre) va a tener y qué tipo de información va a almacenar, en definitiva, debemos **declarar la variable**.

```
int nmAlumnos = 15;  
double radio = 3.14, importe = 102.95;
```

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves. Y lo hacemos indicando su identificador y el tipo de dato, separadas por comas si vamos a declarar varias a la vez, por ejemplo:

De esta forma, estamos declarando `numAlumnos` como una variable de tipo `int`, y otras dos variables `radio` e `importe` de tipo `double`. Aunque no es obligatorio, hemos aprovechado la declaración de las variables para inicializarlas a los valores 15, 3.14 y 102.95 respectivamente.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos como `constante`, utilizando la palabra reservada `final` de la siguiente forma:

```
final double PI = 3.1415926536;
```

En ocasiones puede que al declarar una variable no le demos valor, ¿qué crees que ocurre en estos casos? Pues que el compilador le asigna un valor por defecto, aunque depende del tipo de variable que se trate:

- ✓ Las **variables miembro** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a 0, si son de tipo carácter, se inicializan al carácter null (\0), si son de tipo boolean se les asigna el valor por defecto `false`, y si son tipo referenciado se inicializan a `null`.
- ✓ Las **variables locales** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error. Por ejemplo en este caso:

```
int p;  
  
int q = p; // error
```

Y también en este otro, ya que se intenta usar una variable local que podría no haberse inicializado:

```
int p;  
  
if (. . .)  
    p = 5 ;  
  
int q = p; // error
```

En el ejemplo anterior la instrucción `if` hace que si se cumple la condición que hay entre paréntesis (cualquiera que indiquemos), entonces el programa asignará el valor 5 a la variable `p`; sino se cumple la condición, `p` tomará el valor de `p`. Pero si `p` no se ha inicializado, no tendría valor para asignárselo a `q`. Por ello, el compilador detecta ese posible problema y produce un error del tipo “**La variable podría no haber sido inicializada**”, independientemente de si se cumple o no la condición del `if`.

### Autoevaluación

De las siguientes, señala cuál es la afirmación correcta:

- La declaración de una variable consiste básicamente en indicar el tipo que va a tener seguido del nombre y su valor.
- Java no tiene restricción de tipos.
- Todos los tipos tienen las mismas operaciones a realizar con ellos: suma, resta, multiplicación, etc.
- Todas las anteriores son incorrectas.

Incorrecto, la inicialización de una variable puede ser posterior a la declaración.

Incorrecta, Java es un lenguaje fuertemente tipado.

Incorrecta, las operaciones pueden ser distintas para cada tipo de dato.

¡Exacto! Nada de lo afirmado sobre declaración de variables, tipado de datos y operaciones es correcto.

## Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

### 3.3.- Tipos referenciados.

A partir de los ocho tipos datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman **tipos referenciados** o **referencias**, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
int[] arrayDeEnteros;  
  
Cuenta cuentaCliente;
```



M.ª Flor Moncada Arón (Elaboración Propia)  
(CC BY-NC)

En la primera instrucción declaramos una lista de números del mismo tipo, en este caso, enteros. En la segunda instrucción estamos declarando la variable u objeto `cuentaCliente` como una referencia de tipo `Cuenta`.

Como comentábamos al principio del apartado de Tipos de datos, cualquier aplicación de hoy en día necesita no perder de vista una cierta cantidad de datos. Cuando el conjunto de datos utilizado tiene características similares se suelen agrupar en estructuras para facilitar el acceso a los mismos, son los llamados **datos estructurados**. Son datos estructurados los **arrays, listas, árboles**, etc. Pueden estar en la memoria del programa en ejecución, guardados en el disco como ficheros, o almacenados en una base de datos. Trabajaremos en profundidad con datos estructurados en unidades posteriores.

Además de los ocho tipos de datos primitivos que ya hemos descrito, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato `String`. Java crea automáticamente un nuevo objeto de tipo `String` cuando se encuentra una cadena de caracteres encerrada entre comillas dobles. En realidad se trata de objetos, y por tanto son tipos referenciados, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos:

```
String mensaje;  
  
mensaje= "El primer programa";
```

Hemos visto qué son las variables, cómo se declaran y los tipos de datos que pueden adoptar. Anteriormente hemos visto un ejemplo de creación de variables, en esta ocasión vamos a crear más variables, pero de distintos tipos primitivos y los vamos a mostrar por pantalla. Los tipos referenciados los veremos en la siguiente unidad.

Para mostrar por pantalla un mensaje utilizamos `System.out`, conocido como la salida estándar del programa. Este método lo que hace es escribir un conjunto de caracteres a través de la línea de comandos. En Netbeans esta línea de comandos aparece en la parte inferior de la pantalla. Podemos utilizar `System.out.print` o `System.out.println`. En el segundo caso lo que hace el método es que justo después de escribir el mensaje, sitúa el cursor al principio de la línea siguiente. El texto en color gris que aparece entre caracteres // son comentarios que permiten documentar el código, pero no son tenidos en cuenta por el compilador y, por tanto, no afectan a la ejecución del programa.



M.ª Flor Moncada Arón (Elaboración propia)  
(CC BY-NC)

[Ejemplo de Tipos de Datos en Java](#) (13.00 KB)

#### Debes conocer

Los entornos de desarrollo como Netbeans incluyen multitud de funcionalidades que ayudan al programador a escribir código así como para validar y cambiarlo. Observa las siguientes imágenes:

```

.7
.8     public static void main(String[] args) {
.9         // TODO code application logic here
.10        int i= 10;
.11        double d=3.24;
.12
.13        cannot find symbol
.14            symbol: class string
.15            location: class EjemplosTipos
.16            true;
.17
.18            (Alt-Enter shows hints)
.19
.20
.21
.22
.23
.24
.25

```

Ministerio de Educación y FP ([CC BY-NC](#))

Como ya hemos comprobado en la unidad anterior, cuando escribimos código Java en Netbeans, el editor detecta automáticamente los errores sintácticos y los subraya con color rojo y coloca un pequeño círculo en el número de línea que contiene el error. **Si colocas el ratón sobre ese círculo rojo Netbeans te informará sobre el error detectado.** En el ejemplo, no encuentra la clase **string** porque no existe, su nombre real es **String**. Esto nos facilita mucho la escritura de código porque no se acumulan errores pues los vamos corrigiendo a medida que avanzamos.

Otra de las grandes ventajas de los entornos de desarrollo son las sugerencias de inserción de código y la generación automática de código. A medida que escribes código Java, Netbeans te puede facilitar la inserción generando código automáticamente o sugiriéndote qué escribir en un determinado punto del código pulsando las teclas **Ctrl + Espacio**.

Es muy útil por ejemplo, escribir los primeros caracteres de una palabra reservada y pulsar **Ctrl + Espacio**. Observa la siguiente imagen:

The screenshot shows the NetBeans IDE interface. In the code editor, line 21 contains the start of a 'do' loop: 'do|'. A completion dropdown menu is open, listing suggestions: 'do { ... } dowhile', 'do', 'double', and 'Double'. Below the dropdown, a tooltip says 'Abbreviation: dowhile [TAB for expansion]'. The 'ejemplost' project is selected in the left sidebar, and the 'Output - Ejemplo' window shows build logs for 'ant -f', 'init:', 'deps-c', 'Updating property file /Users/carlosgonzalez/NetBeansProjects/EjemplosTipos/build/buil...', 'Delete', 'clean:', and 'BUILD SUCCESSFUL (total time: 0 seconds)'.

Ministerio de Educación y FP ([CC BY-NC](#))

En el ejemplo, se escribes en el editor los caracteres "do" y se activan las sugerencias con **Ctrl + Espacio**. Netbeans muestra sugerencias, en concreto, tipo primitivo **double**, estructura repetitiva **do** o clase **Double**. Si seleccionas algunas de ellas Netbeans incluirá automáticamente el código Java.

## 3.4.- Tipos enumerados.

Los **tipos de datos enumerados** son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc. Es como si definiéramos nuestro propio tipo de datos.

La forma de declararlos es con la palabra reservada `enum`, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

La lista de valores se coloca entre llaves, porque un tipo de datos `enum` no es otra cosa que una especie de clase en Java, y todas las clases llevan su contenido entre llaves.

Al considerar Java este tipo de datos como si de una clase se tratara, no sólo podemos definir los valores de un tipo enumerado, sino que también podemos definir operaciones a realizar con él y otro tipo de elementos, lo que hace que este tipo de dato sea más versátil y potente que en otros lenguajes de programación.

### Citas para pensar

Oigo y olvido. Veo y recuerdo. Hago y comprendo. *Proverbio chino.*

En el siguiente ejemplo puedes comprobar el uso que se hace de los tipos de datos enumerados. Tenemos una variable `Días` que almacena los días de la semana. Para acceder a cada elemento del tipo enumerado se utiliza el nombre de la variable seguido de un punto y el valor en la lista. Más tarde veremos que podemos añadir métodos y campos o variables en la declaración del tipo enumerado, ya que como hemos comentado un tipo enumerado en Java tiene el mismo tratamiento que las clases.



Mª Flor Moncada Arjón (Elaboración propia)  
(Dominio público)

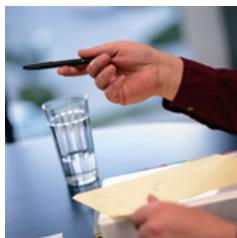
[Tipos de Datos Enumerados en Java](#) (14.00 KB)

En este ejemplo hemos utilizado el método `System.out.print`. Como podrás comprobar si lo ejecutas, la instrucción número 18 escribe el texto que tiene entre comillas pero no salta a la siguiente línea, por lo que el la instrucción número 19 escribe justo a continuación.

Sin embargo, también podemos escribir varias líneas usando una única sentencia. Así lo hacemos en la instrucción número 20, la cual imprime como resultado tres líneas de texto. Para ello hemos utilizado un carácter especial, llamado **carácter escape (\)**. Este carácter sirve para darle ciertas órdenes al compilador, en lugar de que salga impreso en pantalla. Después del carácter de escape viene otro carácter que indica la orden a realizar, juntos reciben el nombre de **secuencia de escape**. La secuencia de escape `\n` recibe el nombre de **carácter de nueva línea**. Cada vez que el compilador se encuentra en un texto ese carácter, el resultado es que mueve el cursor al principio de la línea siguiente. En el próximo apartado vamos a ver algunas de las secuencias de escape más utilizadas.

## 4.- Literales de los tipos primitivos.

### Caso práctico



Stockbyte CD-DVD Num. V43 [\(CC BY-NC\)](#)

**Ada** se encuentra con **María** y **Juan**.

—¿Cómo van esos avances con Java? —**Juan** sabe lo que significa eso, **Ada** se interesa por el trabajo que están llevando a cabo. Ya tienen claro qué tipos de datos utilizar, pero necesitan cerciorarse de los valores que pueden almacenar esos tipos de datos, es decir, qué literales pueden contener, para estar seguros que han hecho la elección adecuada.

—Muy bien —contesta **Juan**. —Si quieres te hacemos una demostración para que veas la estructura del programa.

A **Ada** le satisface la eficacia con que trabajan **María** y **Juan**, apenas ha comenzado con el proyecto y pronto podrá ver resultados inmediatos.

Un **literal**, **valor literal** o **constante literal** es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo `String` o el tipo `null`.

Los **literales booleanos** tienen dos únicos valores que puede aceptar el tipo: `true` y `false`. Por ejemplo, con la instrucción `boolean encontrado = true;` estamos declarando una variable de tipo booleana a la cual le asignamos el valor literal `true`.

Los **literales enteros** se pueden representar en tres notaciones:

- ✓ **Decimal:** por ejemplo `20`. Es la forma más común.
- ✓ **Octal:** por ejemplo `024`. Un número en octal siempre empieza por cero, seguido de dígitos octales (del 0 al 7).
- ✓ **Hexadecimal:** por ejemplo `0x14`. Un número en hexadecimal siempre empieza por `0x` seguido de dígitos hexadecimales (del 0 al 9, de la ‘a’ a la ‘f’ o de la ‘A’ a la ‘F’).

Las constantes literales de tipo `long` se le debe añadir detrás una `L` ó `L`, por ejemplo `873L`, si no se considera por defecto de tipo `int`. Se suele utilizar `L` para evitar la confusión de la ele minúscula con 1.

Los **literales reales** o en coma flotante se expresan con coma decimal o en notación científica, o sea, seguidos de un exponente `e` ó `E`. El valor puede finalizarse con una `f` o una `F` para indicar el formato `float` o con una `d` o una `D` para indicar el formato `double` (por defecto es `double`). Por ejemplo, podemos representar un mismo literal real de las siguientes formas: `13.2D`, `1.32e1`, `0.132E2`. Otras constantes literales reales son por ejemplo: `.54, 31.21E-5, 2.f, 6.022137e+23f, 3.141e-9d`.

Un **literal carácter** puede escribirse como un carácter entre comillas simples como `'a'`, `'ñ'`, `'Z'`, `'p'`, etc. o por su código de la tabla Unicode, anteponiendo la secuencia de escape `\` si el valor lo ponemos en octal o `\u` si ponemos el valor en hexadecimal. Por ejemplo, si sabemos que tanto en `ASCII` como en `Unicode`, la letra A (mayúscula) es el símbolo número 65, y que 65 en octal es 101 y 41 en hexadecimal, podemos representar esta letra como `\101` en octal y `\u0041` en hexadecimal. Existen unos caracteres especiales que se representan utilizando secuencias de escape:

#### Secuencias de escape en Java

Secuencia de escape	Significado	Secuencia de escape	Significado
<code>\b</code>	Retroceso	<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulador	<code>\\"</code>	Carácter comillas dobles
<code>\n</code>	Salto de línea	<code>\'</code>	Carácter comillas simples
<code>\f</code>	Salto de página	<code>\\\</code>	Barra diagonal

Normalmente, los objetos en Java deben ser creados con la orden `new`. Sin embargo, los literales `String` no lo necesitan ya que son objetos que se crean implícitamente por Java.

Los **literales de cadenas de caracteres** se indican entre comillas dobles. En el ejemplo anterior “El primer programa” es un literal de tipo cadena de caracteres. Al construir una cadena de caracteres se puede incluir cualquier carácter Unicode excepto un carácter de retorno de carro, por ejemplo en la siguiente instrucción utilizamos la secuencia de escape \” para escribir dobles comillas dentro del mensaje:

```
String texto = "Juan dijo: \"Hoy hace un dia fantástico...\"";
```

En el ejemplo anterior de tipos enumerados ya estábamos utilizando secuencias de escape, para introducir un salto de línea en una cadena de caracteres, utilizando el carácter especial \n.

## 5.- Operadores y expresiones.

### Caso práctico



Photodisc CD-DVD Num. V07 (CC BY-NC)

**María y Juan** tienen definidas las variables y tipos de datos a utilizar en la aplicación. Es el momento de ponerse a realizar los cálculos que permitan manipular esos datos, sumar, restar, multiplicar, dividir y mucho más. En definitiva se trata de llevar los conocimientos matemáticos al terreno de la programación, ver cómo se pueden hacer operaciones aritméticas, lógicas o de comparación en el lenguaje Java. También necesitarán algún operador que permita evaluar una condición y decidir las acciones a realizar en cada caso. Es importante conocer bien cómo el lenguaje evalúa esas expresiones, en definitiva, cuál es la precedencia que tiene cada operador.

Los **operadores** llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores. Los operadores pueden ser unarios, binarios o terciarios, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y devuelven también un tipo de dato primitivo.

Los operadores se combinan con los literales y/o identificadores para formar expresiones. Una **expresión** es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.

El resultado de una expresión puede ser usado como parte de otra expresión o en una sentencia o instrucción. Las expresiones, combinadas con algunas palabras reservadas o por sí mismas, forman las llamadas **sentencias o instrucciones**.

Por ejemplo, pensemos en la siguiente expresión Java:

```
i + 1;
```

Con esta expresión estamos utilizando un operador aritmético para sumarle una cantidad a la variable `i`, pero es necesario indicar al programa qué hacer con el resultado de dicha expresión:

```
suma = i + 1;
```

Que lo almacene en una variable llamada `suma`, por ejemplo. En este caso ya tendríamos una acción completa, es decir, una sentencia o instrucción.

Más ejemplos de sentencias o instrucciones los tenemos en las declaraciones de variables, vistas en apartados anteriores, o en las estructuras básicas del lenguaje como sentencias condicionales o bucles, que veremos en unidades posteriores.

Como curiosidad comentar que las expresiones de asignación, al poder ser usadas como parte de otras asignaciones u operaciones, son consideradas tanto expresiones en sí mismas como sentencias.

### Citas para pensar

Lo que no hemos realizado no es más que lo que todavía no hemos intentado hacer. Alexis de Tocqueville.



mammaoca2008 (CC BY-NC)

## 5.1.- Operadores aritméticos.

Los operadores aritméticos son aquellos operados que combinados con los operandos forman expresiones matemáticas o aritméticas.

## Operadores aritméticos básicos

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	-10	-10
+	Adición	1.2 + 9.3	10.5
-	Sustracción	312.5 - 12.3	300.2
*	Multiplicación	1.7 * 1.2	1.02
/	División (entera o real)	0.5 / 0.2	2.5
%	Resto de la división entera	25 % 3	1



Paul Schadler (CC BY)

El resultado de este tipo de expresiones depende de los operandos que utilicen:

# Resultados de las operaciones aritméticas en Java

Tipo de los operandos	Resultado
Un operando de tipo <b>long</b> y ninguno real ( <b>float</b> O <b>double</b> )	<b>long</b>
Ningún operando de tipo <b>long</b> ni real ( <b>float</b> O <b>double</b> )	<b>int</b>
Al menos un operando de tipo <b>double</b>	<b>double</b>
Al menos un operando de tipo <b>float</b> y ninguno <b>double</b>	<b>float</b>

Otro tipo de operadores aritméticos son los operadores unarios incrementales y decrementales. Producen un resultado del mismo tipo que el operando, y podemos utilizarlos con **notación prefija**, si el operador aparece antes que el operando, o **notación postfija**, si el operador aparece después del operando. En la tabla puedes ver un ejemplo de utilización de cada operador incremental.

## Operadores incrementales en Java

Tipo operador	Expresión Java	
<b>++ (incremental)</b>	Prefixa: $x=3;$ $y=++x;$ $// x vale 4 e y vale 4$	Postfixa: $x=3;$ $y=x++;$ $// x vale 4 e y vale 3$
<b>--(decremental)</b>	5-- // el resultado es 4	

```

[2]:> print strace -e trace=SIGCHLD ./script1
[3]:> sleep 10
[4]:> kill -15 $!
```

M<sup>a</sup> Flor Moncada Añón (Elaboración propia)  
[\(CC BY-NC\)](#)

En el ejemplo vemos un programa básico que utiliza operadores aritméticos. Observa que usamos `System.out.printf` para mostrar por pantalla un texto formateado. El texto entre dobles comillas son los argumentos del método `printf` y si usamos más de uno, se separan con comas. Primero indicamos cómo queremos que salga el texto, y después el texto que queremos mostrar. Fíjate que con el primer `%s` nos estamos refiriendo a una variable de tipo `String`, o sea, a la primera cadena de texto, con el siguiente `%s` a la segunda cadena y con el último `%s` a la tercera. Con `%f` nos referimos a un argumento de

tipo float, etc.

[Operadores aritméticos](#) (14.00 KB)

## 5.2.- Operadores de asignación.

El principal operador de esta categoría es el operador asignación “`=`”, que permite al programa darle un valor a una variable, y ya hemos utilizado varias ocasiones en esta unidad. Además de este operador, Java proporciona otros operadores de asignación combinados con los operadores aritméticos, que permiten abreviar o reducir ciertas expresiones.

Por ejemplo, el operador “`+=`” suma el valor de la expresión a la derecha del operador con el valor de la variable que hay a la izquierda del operador, y almacena el resultado en dicha variable. En la siguiente tabla se muestran todos los operadores de asignación compuestos que podemos utilizar en Java



aldoaldoz (CC BY-SA)

## Operadores de asignación combinados en Java

Operador	Ejemplo en Java	Expresión equivalente
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

Un ejemplo de operadores de asignación combinados lo tenemos a continuación:



M<sup>a</sup> Flor Moncada Añón (Elaboración propia)  
(CC BY-NC)

(CC BY-NC)

Operadores combinados (14.00 KB)

## Para saber más

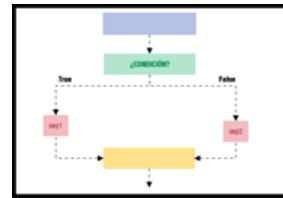
En el siguiente enlace tienes información interesante sobre cómo se pueden utilizar los caracteres especiales incluidos en la orden `printf` (en inglés):

## Orden printf

## 5.3.- Operador condicional.

El operador condicional “?:” sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición. Es el único operador ternario de Java, y como tal, necesita tres operandos para formar una expresión.

El primer operando se sitúa a la izquierda del símbolo de interrogación, y siempre será una expresión booleana, también llamada condición. El siguiente operando se sitúa a la derecha del símbolo de interrogación y antes de los dos puntos, y es el valor que devolverá el operador condicional si la condición es verdadera. El último operando, que aparece después de los dos puntos, es la expresión cuyo resultado se devolverá si la condición evaluada es falsa.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

### Operador condicional en Java

Operador	Expresión en Java
?:	condición ? exp1 : exp2

Por ejemplo, en la expresión:

`(x>y)?x:y;`

Se evalúa la condición de si `x` es mayor que `y`, en caso afirmativo se devuelve el valor de la variable `x`, y en caso contrario se devuelve el valor de `y`.

El operador condicional se puede sustituir por la sentencia `if...then...else` que veremos en la siguiente unidad de Estructuras de control.

### Citas para pensar

**La buena escritura debe ser concisa. Una oración no debe contener palabras innecesarias, un párrafo no debe contener oraciones innecesarias.**

*William Strunk, Jr.*

## 5.4.- Operadores de relación.

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utilizará en otras expresiones o sentencias, que ejecutarán una acción u otra en función de si se cumple o no la relación.

Estas expresiones en Java dan siempre como resultado un valor booleano true o false. En la tabla siguiente aparecen los operadores relacionales en Java.

Operadores relacionales en Java

Operador	Ejemplo en Java	Significado
==	op1 == op2	op1 igual a op2
!=	op1 != op2	op1 distinto de op2
>	op1 > op2	op1 mayor que op2
<	op1 < op2	op1 menor que op2
>=	op1 >= op2	op1 mayor o igual que op2
<=	op1 <= op2	op1 menor o igual que op2



Mª Flor Moncada Arón (Elaboración propia) [CC BY-NC](#)

Hasta ahora hemos visto ejemplos que creaban variables y se inicializaban con algún valor. Pero ¿y si lo que queremos es que el usuario introduzca un valor al programa? Entonces debemos agregarle interactividad a nuestro programa, por ejemplo utilizando la clase `Scanner`. Aunque no hemos visto todavía qué son las clases y los objetos, de momento vamos a pensar que la clase `Scanner` nos va a permitir leer los datos que se escriben por teclado, y que para usarla es necesario importar el paquete de clases que la contiene. El código del ejemplo lo tienes a continuación. El programa se quedará esperando a que el usuario escriba algo en el teclado y pulse la tecla intro. En ese momento se convierte lo leído a un valor del tipo `int` y lo guarda en la variable indicada. Además de los operadores relacionales, en este ejemplo utilizamos también el operador condicional, que compara si los números son iguales. Si lo son, devuelve la cadena iguales y sino, la cadena distintos.

```
public class EjemploRelacionales {
    public static void main(String args[]) {
        // creamos Scanner para petición de datos
        Scanner teclado = new Scanner(System.in);
        int x, y;
        boolean iguales;
        System.out.println("Introduce primer número: ");
        x = teclado.nextInt(); // pedimos el primer número al usuario
        System.out.println("Introduce segundo número: ");
        y = teclado.nextInt(); // pedimos el segundo número al usuario
        // comprobamos las comparaciones
        if (x == y) {
            System.out.println("Los números iguales");
        } else {
            System.out.println("Los números distintos");
        }
    }
}
```

Mª Flor Moncada Arón (Elaboración propia)  
[CC BY-NC](#)

[Operadores de relación y condicional](#) (14.00 KB)

## Autoevaluación

Señala cuáles son los operadores relacionales en Java.

- ==, !=, >, <, >=, <=.
- =, !=, >, <, >=, <=.
- ==, !=, >, <, =>, ==.
- ==, !=, >, <, >=, <=.

No es correcta, porque hay un error de sintaxis al colocar un espacio entre los símbolos de cada operador.

Incorrecta, porque = es el operador de asignación, no el operador relacional de igualdad ==.

No es la respuesta correcta, porque los símbolos de los operadores distinto a, mayor o igual y menor o igual son !=, >= y <=, respectivamente.

Muy bien. Tienes claro cuáles son los operadores relacionales en Java.

## Solución

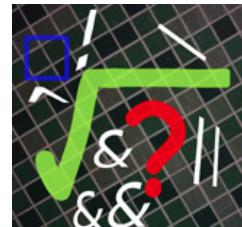
1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

## 5.5.- Operadores lógicos.

Los operadores lógicos realizan operaciones sobre valores booleanos, o resultados de expresiones relacionales, dando como resultado un valor booleano.

Los operadores lógicos los podemos ver en la tabla que se muestra a continuación. Existen ciertos casos en los que el segundo operando de una expresión lógica no se evalúa para ahorrar tiempo de ejecución, porque con el primero ya es suficiente para saber cuál va a ser el resultado de la expresión.

Por ejemplo, en la expresión `a && b` si `a` es falso, no se sigue comprobando la expresión, puesto que ya se sabe que la condición de que ambos sean verdadero no se va a cumplir. En estos casos es más conveniente colocar el operando más propenso a ser falso en el lado de la izquierda. Igual ocurre con el operador `||`, en cuyo caso es más favorable colocar el operando más propenso a ser verdadero en el lado izquierdo.



aldoaldoz (CC BY-NC-SA)

### Operadores lógicos en Java

Operador	Ejemplo en Java	Significado
!	<code>!op</code>	Devuelve true si el operando es false y viceversa.
&	<code>op1 &amp; op2</code>	Devuelve true si op1 y op2 son true
	<code>op1   op2</code>	Devuelve true si op1 u op2 son true
^	<code>op1 ^ op2</code>	Devuelve true si sólo uno de los operandos es true
&&	<code>op1 &amp;&amp; op2</code>	Igual que &, pero si op1 es false ya no se evalúa op2
	<code>op1    op2</code>	Igual que  , pero si op1 es true ya no se evalúa op2

En el siguiente código puedes ver un ejemplo de utilización de operadores lógicos:



Má Flor Moncada Arón (Elaboración propia)  
[\(CC BY-NC\)](#)

[Operadores lógicos](#) (14.00 KB)

## 5.6.- Operadores de bits.

Los operadores a nivel de bits se caracterizan porque realizan operaciones sobre números enteros (o `char`) en su representación binaria, es decir, sobre cada dígito binario.

En la tabla tienes los operadores a nivel de bits que utiliza Java.

**Operadores a nivel de bits en Java**

Operador	Ejemplo en Java	Significado
<code>~</code>	<code>~op</code>	Realiza el complemento binario de op (invierte el valor de cada bit)
<code>&amp;</code>	<code>op1 &amp; op2</code>	Realiza la operación AND binaria sobre op1 y op2
<code> </code>	<code>op1   op2</code>	Realiza la operación OR binaria sobre op1 y op2
<code>^</code>	<code>op1 ^ op2</code>	Realiza la operación OR-exclusivo (XOR) binaria sobre op1 y op2
<code>&lt;&lt;</code>	<code>op1 &lt;&lt; op2</code>	Desplaza op2 veces hacia la izquierda los bits de op1
<code>&gt;&gt;</code>	<code>op1 &gt;&gt; op2</code>	Desplaza op2 veces hacia la derecha los bits de op1
<code>&gt;&gt;&gt;</code>	<code>op1 &gt;&gt;&gt; op2</code>	Desplaza op2 (en positivo) veces hacia la derecha los bits de op1



Mª Flor Moncada Añón (Elaboración propia)  
[\(CC BY-NC\)](#)

### Para saber más

Los operadores de bits raramente los vas a utilizar en tus aplicaciones de gestión. No obstante, si sientes curiosidad sobre su funcionamiento, puedes ver el siguiente enlace dedicado a este tipo de operadores:

[Operadores de bits](#)

## 5.7.- Trabajo con cadenas.

Ya hemos visto en el apartado de literales que el objeto `String` se corresponde con una secuencia de caracteres entrecomillados, como por ejemplo “`hola`”. Este literal se puede utilizar en Java como si de un tipo de datos primitivo se tratase, y, como caso especial, no necesita la orden `new` para ser creado.

No se trata aquí de que nos adentremos en lo que es una clase u objeto, puesto que lo veremos en unidades posteriores, y trabajaremos mucho sobre ello. Aquí sólo vamos a utilizar determinadas operaciones que podemos realizar con el objeto `String`, y lo verás mucho más claro con ejemplos descriptivos.

Para aplicar una operación a una variable de tipo `String`, escribiremos su nombre seguido de la operación, separados por un punto. Entre las principales operaciones que podemos utilizar para trabajar con cadenas de caracteres están las siguientes:

- ✓ **Creación.** Como hemos visto en el apartado de literales, podemos crear una variable de tipo `String` simplemente asignándole una cadena de caracteres encerrada entre comillas dobles.
- ✓ **Obtención de longitud.** Si necesitamos saber la longitud de un `String`, utilizaremos el método `length()`.
- ✓ **Concatenación.** Se utiliza el operador `+` o el método `concat()` para concatenar cadenas de caracteres.
- ✓ **Comparación.** El método `equals()` nos devuelve un valor booleano que indica si las cadenas comparadas son o no iguales. El método `equalsIgnoreCase()` hace lo propio, ignorando las mayúsculas de las cadenas a considerar.
- ✓ **Obtención de subcadenas.** Podemos obtener cadenas derivadas de una cadena original con el método `substring()`, al cual le debemos indicar el inicio y el fin de la subcadena a obtener.
- ✓ **Cambio a mayúsculas/minúsculas.** Los métodos `toUpperCase()` y `toLowerCase()` devuelven una nueva variable que transforma en mayúsculas o minúsculas, respectivamente, la variable inicial.
- ✓ **Valueof.** Utilizaremos este método para convertir un tipo de dato primitivo (`int`, `long`, `float`, etc.) a una variable de tipo `String`.

A continuación varios ejemplos de las distintas operaciones que podemos realizar concadenas de caracteres o `String` en Java:



Mª Flor Moncada Añón (Elaboración propia)

[\(CC BY-NC\)](#)

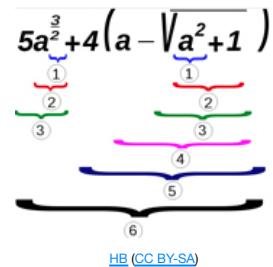
[Operaciones con cadenas](#) (14.00 KB)

## 5.8.- Precedencia de operadores.

El orden de precedencia de los operadores determina la secuencia en que deben realizarse las operaciones cuando en una expresión intervienen operadores de distinto tipo.

Las reglas de precedencia de operadores que utiliza Java coinciden con las reglas de las expresiones del álgebra convencional. Por ejemplo:

- ✓ La multiplicación, división y resto de una operación se evalúan primero. Si dentro de la misma expresión tengo varias operaciones de este tipo, empezaré evaluándolas de izquierda a derecha.
- ✓ La suma y la resta se aplican después que las anteriores. De la misma forma, si dentro de la misma expresión tengo varias sumas y restas empezaré evaluándolas de izquierda a derecha.



A la hora de evaluar una expresión es necesario tener en cuenta la **asociatividad** de los operadores. La asociatividad indica qué operador se evalúa antes, en condiciones de igualdad de precedencia. Los operadores de asignación, el operador condicional (`:=`), los operadores incrementales (`++`, `--`) y el casting son asociativos por la derecha. El resto de operadores son asociativos por la izquierda, es decir, que se empiezan a calcular en el mismo orden en el que están escritos: de izquierda a derecha. Por ejemplo, en la expresión siguiente:

`10 / 2 * 5;`

Realmente la operación que se realiza es  $(10 / 2) * 5$ , porque ambos operadores, división y multiplicación, tienen igual precedencia y por tanto se evalúa primero el que antes nos encontramos por la izquierda, que es la división. El resultado de la expresión es 25. Si fueran asociativos por la derecha, puedes comprobar que el resultado sería diferente, primero multiplicaríamos  $2 * 5$  y luego dividiríamos entre 10, por lo que el resultado sería 1. En esta otra expresión:

`x = y = z = 1;`

Realmente la operación que se realiza es  $x = (y = (z = 1))$ . Primero asignamos el valor de 1 a la variable `z`, luego a la variable `y`, para terminar asignando el resultado de esta última asignación a `x`. Si el operador asignación fuera asociativo por la izquierda esta operación no se podría realizar, ya que intentaríamos asignar a `x` el valor de `y`, pero `y` aún no habría sido inicializada.

En la tabla se detalla el orden de precedencia y la asociatividad de los operadores que hemos comentado en este apartado. Los operadores se muestran de mayor a menor precedencia.

**Orden de precedencia de operadores en Java**

Operador	Tipo	Asociatividad
<code>++ --</code>	Unario, notación postfija	Derecha
<code>++ -- ++</code> (cast) <code>! ~</code>	Unario, notación prefija	Derecha
<code>* / %</code>	Aritméticos	Izquierda
<code>+ -</code>	Aritméticos	Izquierda
<code>&lt;&lt;&gt;&gt;&gt;</code>	Bits	Izquierda
<code>&lt;&lt;=&gt;</code>	Relacionales	Izquierda
<code>== !=</code>	Relacionales	Izquierda
<code>&amp;</code>	Lógico, Bits	Izquierda
<code>^</code>	Lógico, Bits	Izquierda

Operador	Tipo	Asociatividad
I	Lógico, Bits	Izquierda
&&	Lógico	Izquierda
	Lógico	Izquierda
?:	Operador condicional	Derecha
= += -= *= /= %=	Asignación	Derecha

## Reflexiona

¿Crees que es una buena práctica de programación utilizar paréntesis en expresiones aritméticas complejas, aún cuando no sean necesarios?

[Mostrar retroalimentación](#)

Probablemente acertaste. El uso de paréntesis, incluso cuando no son necesarios, puede hacer más fácil de leer las expresiones aritméticas complejas.

## 6.- Conversion de tipo.

### Caso práctico



Stockbyte CD-DVD Num. V43 [CC BY-NC](#)

**María** ha avanzado mucho en sus conocimientos sobre Java y ha contado con mucha ayuda por parte de **Juan**. Ahora mismo tiene un problema con el código, y le comenta —Estoy atrancada en el código. Tengo una variable de tipo byte y quiero asignarle un valor de tipo int, pero el compilador me da un error de posible pérdida de precisión ¿tú sabes qué significa eso?. —Claro —le contesta Juan, —es un problema de conversión de tipos, para asignarle el valor a la variable de tipo byte debes hacer un casting. —¡Ah! —dice María, —¿y cómo se hace eso?

Imagina que queremos dividir un número entre otro ¿tendrá decimales el resultado de esa división? Podemos pensar que siempre que el denominador no sea divisible entre el divisor, tendremos un resultado con decimales, pero no es así. Si el denominador y el divisor son variables de tipo entero, el resultado será entero y no tendrá decimales. Para que el resultado tenga decimales necesitaremos hacer una **conversión de tipo**.

Las conversiones de tipo se realizan para hacer que el resultado de una expresión sea del tipo que nosotros deseamos, en el ejemplo anterior para hacer que el resultado de la división sea de tipo real y, por ende, tenga decimales. Existen dos tipos de conversiones:

- ✓ **Conversiones automáticas.** Cuando a una variable de un tipo se le asigna un valor de otro tipo numérico con menos bits para su representación, se realiza una conversión automática. En ese caso, el valor se dice que es promocionado al tipo más grande (el de la variable), para poder hacer la asignación. También se realizan conversiones automáticas en las operaciones aritméticas, cuando estamos utilizando valores de distinto tipo, el valor más pequeño se promociona al valor más grande, ya que el tipo mayor siempre podrá representar cualquier valor del tipo menor (por ejemplo, de `int` a `long` o de `float` a `double`).
- ✓ **Conversiones explícitas.** Cuando hacemos una conversión de un tipo con más bits a un tipo con menos bits. En estos casos debemos indicar que queremos hacer la conversión de manera expresa, ya que se puede producir una pérdida de datos y hemos de ser conscientes de ello. Este tipo de conversiones se realiza con el **operador cast**. El operador `cast` es un operador unario que se forma colocando delante del valor a convertir el tipo de dato entre paréntesis. Tiene la misma precedencia que el resto de operadores unarios y se asocia de izquierda a derecha.

Deberemos tener en cuenta que **un valor numérico nunca puede ser asignado a una variable de un tipo menor en rango, si no es con una conversión explícita**. Por ejemplo:

```
int a;  
  
byte b;  
  
a = 12;           // no se realiza conversión alguna  
  
b = 12;           // se permite porque 12 está dentro del rango permitido de valores para b  
  
b = a;            // error, no permitido (incluso aunque 12 podría almacenarse en un byte)  
  
byte b = (byte) a; // Correcto, forzamos conversión explícita
```

En el ejemplo anterior vemos un caso típico de error de tipos, ya que estamos intentando asignarle a `b` el valor de `a`, siendo `b` de un tipo más pequeño. Lo correcto es promocionar `a` al tipo de datos `byte`, y entonces asignarle su valor a la variable `b`.

## Debes conocer

En el [anexo I](#) tienes información adicional sobre la conversión de tipos en el lenguaje Java.

## 7.- Comentarios.

### Caso práctico



Juan ha podido comprobar los avances que ha hecho María con la programación. Ya domina todos los aspectos básicos sobre sintaxis, estructura de un programa, variables y tipos de datos. Ada le acaba de comunicar que van a sumarse al proyecto dos personas más, Ana y Carlos que están haciendo las prácticas del ciclo de Desarrollo de Aplicaciones Multiplataforma en la empresa. —Al principio de cada programa indicaremos una breve descripción y el autor. En operaciones complicadas podríamos añadir un comentario les ayudará a entender mejor qué es lo que hace —indica Juan. —De acuerdo —comenta María, —y podemos ir metiendo los comentarios de la herramienta esa que me comentaste, Javadoc, para que se cree una documentación aún más completa. — ¡Ajá! —asiente Juan— pues ¡ manos a la obra!

Los comentarios son muy importantes a la hora de describir qué hace un determinado programa. A lo largo de la unidad los hemos utilizado para documentar los ejemplos y mejorar la comprensión del código. Para lograr ese objetivo, es normal que cada programa comience con unas líneas de comentario que indiquen, al menos, una breve descripción del programa, el autor del mismo y la última fecha en que se ha modificado.

Todos los lenguajes de programación disponen de alguna forma de introducir comentarios en el código. En el caso de Java, nos podemos encontrar los siguientes tipos de comentarios:

- ✓ **Comentarios de una sola línea.** Utilizaremos el delimitador // para introducir comentarios de sólo una línea.

```
// comentario de una sola línea
```

- ✓ **Comentarios de múltiples líneas.** Para introducir este tipo de comentarios, utilizaremos una barra inclinada y un asterisco (\*), al principio del párrafo y un asterisco seguido de una barra inclinada (\*) al final del mismo.

```
/* Esto es un comentario  
de varias líneas */
```

- ✓ **Comentarios Javadoc.** Utilizaremos los delimitadores /\*\* y \*\*/. Al igual que con los comentarios tradicionales, el texto entre estos delimitadores será ignorado por el compilador. Este tipo de comentarios se emplean para generar documentación automática del programa. A través del programa javadoc, incluido en JavaSE , se recogen todos estos comentarios y se llevan a un documento en formato .html.

```
/** Comentario de documentación.  
  
Javadoc extrae los comentarios del código y  
genera un archivo html a partir de este tipo de comentarios  
*/
```

### Reflexiona

Una buena práctica de programación es añadir en la última llave que delimita cada bloque de código, un

comentario indicando a qué clase o método pertenece esa llave.

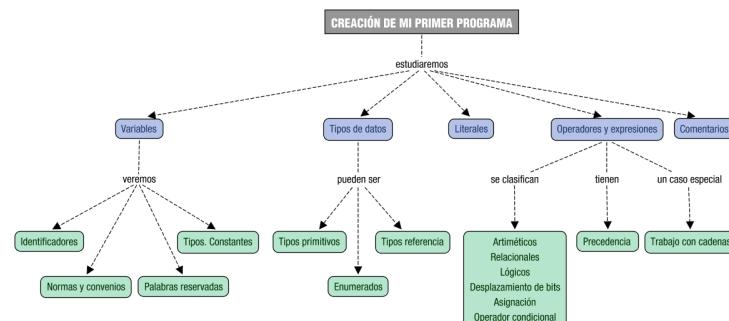
## Para saber más

Si quieres ir familiarizándote con la información que hay en la web de Oracle, en el siguiente enlace puedes encontrar más información sobre la herramienta Javadoc incluida en el Kit de Desarrollo de Java SE (en inglés):

[Página oficial de Oracle sobre la herramienta Javadoc](#)

## 8.- Conclusiones.

Durante el desarrollo de esta unidad hemos trabajado con los diferentes tipos de datos del lenguaje Java, los diferentes tipos de variables existentes y los diferentes operadores que permiten la realización de operaciones con datos. Además, hemos aprendido a declarar los diferentes tipos de variables y a utilizar un amplio rango de operadores, analizando la precedencia en el uso de los mismos. En el mapa conceptual se pueden observar los conceptos que hay que tener claros antes de abordar la siguiente unidad:



Ministerio de Educación y FP ([CC BY-NC](#))

En la siguiente unidad empezaremos a trabajar con los conceptos de clase y objeto, dos de los pilares básicos del paradigma de programación orientado a objetos.

# Anexo I.- Conversión de tipos de datos en Java.

Tabla de Conversión de Tipos de Datos Primitivos

		Tipo destino								
		boolean	char	byte	short	int	long	float	double	
Tipo origen	boolean	-	N	N	N	N	N	N	N	
	char	N	-	C	C	CI	CI	CI	CI	
	byte	N	C	-	CI	CI	CI	CI	CI	
	short	N	C	C	-	CI	CI	CI	CI	
	int	N	C	C	C	-	CI	CI*	CI	
	long	N	C	C	C	C	-	CI*	CI*	
	float	N	C	C	C	C	C	-	CI	
	double	N	C	C	C	C	C	C	-	

Explicación de los símbolos utilizados:

**N:** Conversión no permitida (un `boolean` no se puede convertir a ningún otro tipo y viceversa).

**CI:** Conversión implícita o automática. Un asterisco indica que puede haber posible pérdida de datos.

**C:** Casting de tipos o conversión explícita.

El asterisco indica que puede haber una posible pérdida de datos, por ejemplo al convertir un número de tipo `int` que usa los 32 bits posibles de la representación, a un tipo `float`, que también usa 32 bits para la representación, pero 8 de los cuales son para el exponente.

En cualquier caso, las conversiones de números en coma flotante a números enteros siempre necesitarán un **Casting**, y deberemos tener mucho cuidado debido a la pérdida de precisión que ello supone.

## Reglas de Promoción de Tipos de Datos

Cuando en una expresión hay datos o variables de distinto tipo, el compilador realiza la promoción de unos tipos en otros, para obtener como resultado el tipo final de la expresión. Esta promoción de tipos se hace siguiendo unas reglas básicas en base a las cuales se realiza esta promoción de tipos, y resumidamente son las siguientes:

- ✓ Si uno de los operandos es de tipo `double`, el otro es convertido a `double`.
- ✓ En cualquier otro caso:
  - ◆ Si el uno de los operandos es `float`, el otro se convierte a `float`
  - ◆ Si uno de los operandos es `long`, el otro se convierte a `long`
  - ◆ Si no se cumple ninguna de las condiciones anteriores, entonces ambos operandos son convertidos al tipo `int`.

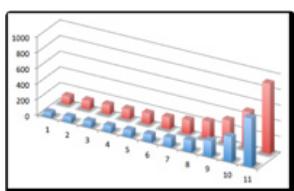
Tabla sobre otras consideraciones con los Tipos de Datos

Conversiones de números en Coma flotante ( <code>float</code> , <code>double</code> ) a enteros ( <code>int</code> )	Conversiones entre caracteres ( <code>char</code> ) y enteros ( <code>int</code> )	Conversiones de tipos de caracteres
<p>Cuando convertimos números en coma flotante a números enteros, la parte decimal se trunca (redondeo a cero). Si queremos hacer otro tipo de redondeo, podemos utilizar, entre otras, las siguientes funciones:</p> <ul style="list-style-type: none"> <li>✓ <b>Math.round(num):</b> Redondeo al siguiente número entero.</li> <li>✓ <b>Math.ceil(num):</b> Mínimo entero que sea mayor o igual a num.</li> </ul>	<p>Como un tipo <code>char</code> lo que guarda en realidad es el código <b>Unicode</b> de un carácter, los caracteres pueden ser considerados como números enteros sin signo.</p> <p><b>Ejemplo:</b></p> <pre>int num;</pre>	<p>Para convertir cadenas otros tipos de datos siguientes funciones:</p> <pre>num=Byte.parseByte(); num=Short.parseShort();</pre>

Conversiones de números en Coma flotante (float, double) a enteros (int)	Conversiones entre caracteres (char) y enteros (int)	Conversiones de tipos de caracteres
<p>✓ <b>Math.floor(num):</b> Entero mayor, que sea inferior o igual a num.</p> <pre>double num=3.5; x=Math.round(num);      // x = 4 y=Math.ceil(num);       // y = 4 z=Math.floor(num);      // z = 3</pre>	<pre>char c; num = (int) 'A';           //num = 65 c = (char) 65;             // c = 'A' c = (char) ((int) 'A' + 1); // c = 'B'</pre>	<pre>num=Integer.parseInt(cadena); num=Long.parseLong(cadena); num=Float.parseFloat(cadena); num=Double.parseDouble(cadena);  Por ejemplo, si he teculado un número almacenado en una String llamada cadena, convertir al tipo d haríamos lo siguiente:</pre> <pre>byte n=Byte.parseByte(cadena);</pre>

# Utilización de objetos.

## Caso práctico



[f.e.weaver \(CC BY\)](#)

**Ada y Juan** se han reunido para discutir sobre distintos proyectos de **BK Programación**. Ada le comenta a Juan que están teniendo algunos problemas con determinados proyectos. A menudo surgen modificaciones o mejoras en el software en el ámbito de los contratos de mantenimiento que tienen suscritos con los clientes, y realizar las modificaciones en los programas está suponiendo en muchos casos modificar el programa casi en su totalidad.

A eso se ha de sumar que las tareas de modificación son encargadas a las personas más adecuadas en ese momento, según la carga de trabajo que haya; que no tienen por qué coincidir con las personas que desarrollaron el programa. Las modificaciones en los proyectos se están retrasando, y hay algunas que deben estar listas antes de que surja el nuevo cambio de versión.

En reuniones anteriores se ha comentado la posibilidad de aumentar el precio del contrato de mantenimiento de los clientes. Se ha consultado con el equipo de comerciales y a regañadientes han aceptado un aumento que aún está por decidir, pero aún así quizás no sea suficiente. La empresa necesita mejorar el método de trabajo para reducir costes de mantenimiento del software y alcanzar la rentabilidad deseada.



[Ministerio de Educación y Formación Profesional. \(Dominio público\)](#)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Introducción.

---



[JoshuaDavisPhotography](#) (CC BY-NC-SA)

Si nos paramos a observar el mundo que nos rodea, podemos apreciar que casi todo está formado por **objetos**. Existen coches, edificios, sillas, mesas, semáforos, ascensores e incluso personas o animales. **Todos ellos pueden ser considerados objetos, con una serie de características y comportamientos.** Por ejemplo, existen coches de diferentes marcas, colores, etc. y pueden acelerar, frenar, girar, etc., o las personas tenemos diferente color de pelo, ojos, altura y peso y podemos nacer, crecer, comer, dormir, etc.

Los programas son el resultado de la búsqueda y obtención de una solución para un problema del mundo real. Pero ¿en qué medida los programas están organizados de la misma manera que el problema que tratan de solucionar? La respuesta es que muchas veces los programas se ajustan más a los términos del sistema en el que se ejecutarán que a los del propio problema.

Si redactamos los programas utilizando los mismos términos de nuestro mundo real, es decir, utilizando objetos, y no los términos del sistema o computadora donde se vaya a ejecutar, conseguiremos que éstos sean más legibles y, por tanto, más fáciles de modificar.

Esto es precisamente lo que pretende la **Programación Orientada a Objetos (POO)**, en inglés **OOP (Object Oriented Programming)**, establecer una serie de técnicas que permitan trasladar los problemas del mundo real a nuestro sistema informático. Ahora que ya conocemos la sintaxis básica de Java, es el momento de comenzar a utilizar las características orientadas a objetos de este lenguaje, y estudiar los conceptos fundamentales de este modelo de programación.

## 2.- Fundamentos de la Programación Orientada a Objetos.

### Caso práctico



PetsitUSA Pet Sitter Directory (CC BY-SA)

Juan cuenta con la ayuda de María para desarrollar la aplicación para la Clínica Veterinaria. Lo normal es pensar en tener una aplicación de escritorio para las altas y bajas de clientes y la gestión de mascotas, y una parte web para que la clínica pueda estar presente en Internet e, incluso, realizar la venta on-line de sus productos. María tiene bastante experiencia en administración de páginas web, pero para estar capacitada en el desarrollo de aplicaciones en Java, necesita adquirir conocimientos adicionales.

Juan le explica que tienen que utilizar un método de programación que les ayude a organizar los programas, a trabajar en equipo de forma que si uno de ellos tiene que dejar una parte para que se encargue el otro, que éste lo pueda retomar con el mínimo esfuerzo. Además, interesa poder reutilizar todo el código que vayan creando, para ir más rápido a la hora de programar. Juan le explica que si consiguen adoptar ese método de trabajo, no sólo redundará en una mejor organización para ellos, sino que ayudará a que las modificaciones en los programas sean más llevaderas de lo que lo están siendo ahora.

María asiente ante las explicaciones de Juan, e intuye que todo lo entenderá mejor conforme vaya conociendo los conceptos de Programación Orientada a Objetos.

De lo que realmente se trata es de que **BK Programación** invierta el menor tiempo posible en los proyectos que realice, aprovechando material elaborado con el esfuerzo ya realizado en otras aplicaciones.

Dentro de las distintas formas de hacer las cosas en programación, distinguimos dos paradigmas fundamentales:

- ✓ **Programación Estructurada**, se crean funciones y procedimientos que definen las acciones a realizar, y que posteriormente forman los programas.
- ✓ **Programación Orientada a Objetos**, considera los programas en términos de **objetos** y todo gira alrededor de ellos.



bellydraft (CC BY)

Pero ¿en qué consisten realmente estos paradigmas? Veamos estos dos modelos de programación con más detenimiento. Inicialmente se programaba aplicando las técnicas de programación tradicional, también conocidas como **Programación Estructurada**. El problema se descomponía en unidades más pequeñas hasta llegar a acciones o verbos muy simples y fáciles de codificar. Por ejemplo, en la resolución de una ecuación de primer grado, lo que hacemos es descomponer el problema en acciones más pequeñas o pasos diferenciados:

- ✓ Pedir valor de los coeficientes.
- ✓ Calcular el valor de la incógnita.
- ✓ Mostrar el resultado.

Si nos damos cuenta, esta serie de acciones o pasos diferenciados no son otra cosa que verbos; por ejemplo el verbo pedir, calcular, mostrar, etc.

Sin embargo, la Programación Orientada a Objetos aplica de otra forma diferente la **técnica de programación "divide y vencerás"**. Este paradigma surge en un intento de salvar las dificultades que, de forma innata, posee el software. Para ello lo que hace es descomponer, en lugar de acciones, en objetos. El principal objetivo sigue siendo descomponer el problema en problemas más pequeños, que sean fáciles de manejar y mantener, fijándonos en cuál es el escenario del problema e intentando reflejarlo en nuestro programa. O sea, se trata de trasladar la visión del mundo real a nuestros programas. Por este motivo se dice que la **Programación Orientada a Objetos aborda los problemas de una forma más natural**, entendiendo como natural que está más en contacto con el mundo que nos rodea.

La **Programación Estructurada** se centra en el conjunto de acciones a realizar en un programa, haciendo una división de procesos y datos. La **Programación Orientada a Objetos** se centra en la relación que existe entre los datos y las acciones a realizar con ellos, y los encierra dentro del concepto de objeto, tratando de realizar una abstracción lo más cercana al mundo real. Sin embargo ambas no son excluyentes: parte de los objetos en el

paradigma orientado a objetos es implementada siguiendo los principio de la programación estructurada.

**La Programación Orientada a Objetos es un sistema o conjunto de reglas que nos ayudan a descomponer la aplicación en objetos. A menudo se trata de representar las entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación. Es decir, debemos establecer una correspondencia directa entre el espacio del problema y el espacio de la solución.** ¿Pero en la práctica esto qué quiere decir? Pues que a la hora de escribir un programa, nos fijaremos en los objetos involucrados, sus características comunes y las acciones que pueden realizar. Una vez localizados los objetos que intervienen en el problema real (espacio del problema), los tendremos que trasladar al programa informático (espacio de la solución). Con este planteamiento, la solución a un problema dado se convierte en una tarea sencilla y bien organizada.

## Autoevaluación

**Relaciona el término con su definición, escribiendo el número asociado a la definición en el hueco correspondiente.**

### Ejercicio de relacionar

Paradigma	Relación	Definición
Programación Orientada a Objetos.	<input type="radio"/>	1. Maneja funciones y procedimientos que definen las acciones a realizar.
Programación Estructurada.	<input type="radio"/>	2. Representa las entidades del mundo real mediante componentes de la aplicación.

Enviar

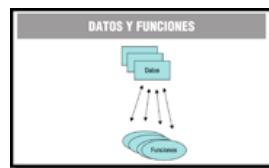
La Programación Orientada a Objetos y la Programación Estructurada son paradigmas o modelos de programación.

## 2.1.- Conceptos.

Para entender mejor la filosofía de orientación a objetos veamos algunas características que la diferencian de las técnicas de programación tradicional.

En la Programación Estructurada, el programa estaba compuesto por un conjunto de **datos y funciones "globales"**. El término global significaba que eran accesibles por todo el programa, pudiendo ser llamados en cualquier ubicación de la aplicación. Dentro de las funciones se situaban las instrucciones del programa que manipulaban los datos. **Funciones y datos se encontraban separados y totalmente independientes**. Esto ocasionaba dos problemas principales:

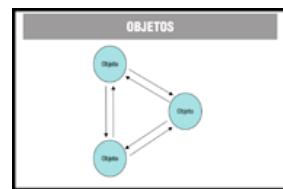
- ✓ Los programas se creaban y estructuraban de acuerdo con la arquitectura de la computadora donde se tenían que ejecutar.
- ✓ Al estar separados los datos de las funciones, éstos eran visibles en toda la aplicación. Ello ocasionaba que cualquier modificación en los datos podía requerir la modificación en todas las funciones del programa, en correspondencia con los cambios en los datos.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

En la **Programación Orientada a Objetos** la situación es diferente. La utilización de **objetos** permite un mayor nivel de **abstracción** que con la Programación Estructurada, y ofrece las siguientes diferencias con respecto a ésta:

- ✓ El programador organiza su programa en **objetos**, que son **representaciones del mundo real** que están más cercanas a la forma de pensar de la gente.
- ✓ Los datos, junto con las funciones que los manipulan, son parte interna de los objetos y no están accesibles al resto de los objetos. Por tanto, los cambios en los datos de un objeto sólo afectan a las funciones definidas para ese objeto, pero no al resto de la aplicación.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

Todos los programas escritos bajo el paradigma orientado a Objetos se pueden escribir igualmente mediante la Programación Estructurada. Sin embargo, la Programación Orientada a Objetos es la que mayor facilidad presenta para el desarrollo de programas basados en interfaces gráficas de usuario.

## 2.2.- Beneficios.

Según lo que hemos visto hasta ahora, **un objeto es cualquier entidad que podemos ver o apreciar**. El concepto fundamental de la Programación Orientada a Objetos son, precisamente, los objetos. Pero ¿qué beneficios aporta la utilización de objetos? Fundamentalmente la posibilidad de representar el problema en términos del mundo real, que como hemos dicho están más cercanos a nuestra forma de pensar, pero existen otra serie de ventajas como las siguientes:

- ✓ **Comprendión.** Los conceptos del espacio del problema se hayan reflejados en el código del programa, por lo que la mera lectura del código nos describe la solución del problema en el mundo real.
- ✓ **Modularidad.** Facilita la modularidad del código, al estar las definiciones de objetos en módulos o archivos independientes, hace que las aplicaciones estén mejor organizadas y sean más fáciles de entender.
- ✓ **Fácil mantenimiento.** Cualquier modificación en las acciones queda automáticamente reflejada en los datos, ya que ambos están estrechamente relacionados. Esto hace que el mantenimiento de las aplicaciones, así como su corrección y modificación sea mucho más fácil. Por ejemplo, podemos querer utilizar un algoritmo más rápido, sin tener que cambiar el programa principal. Por otra parte, al estar las aplicaciones mejor organizadas, es más fácil localizar cualquier elemento que se quiera modificar y/o corregir. Esto es importante ya que se estima que los mayores costes de software no están en el proceso de desarrollo en sí, sino en el mantenimiento posterior de ese software a lo largo de su vida útil.
- ✓ **Seguridad.** La probabilidad de cometer errores se ve reducida, ya que no podemos modificar los datos de un objeto directamente, sino que debemos hacerlo mediante las acciones definidas para ese objeto. Imaginemos un objeto lavadora. Se compone de un motor, tambor, cables, tubos, etc. Para usar una lavadora no se nos ocurre abrirla y empezar a manipular esos elementos, ya que lo más probable es que se estropee. En lugar de eso utilizamos los programas de lavado establecidos. Pues algo parecido con los objetos, no podemos manipularlos internamente, sólo utilizar las acciones que para ellos hay definidas.
- ✓ **Reusabilidad.** Los objetos se definen como entidades reutilizables, es decir, que los programas que trabajan con las mismas estructuras de información, pueden reutilizar las definiciones de objetos empleadas en otros programas, e incluso las acciones definidas sobre ellos. Por ejemplo, podemos crear la definición de un objeto de tipo **persona** para una aplicación de negocios y deseamos construir a continuación otra aplicación, digamos de educación, en donde utilizamos también personas, no es necesario crear de nuevo el objeto, sino que por medio de la reusabilidad podemos utilizar el tipo de objeto **persona** previamente definido.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

### Citas para pensar

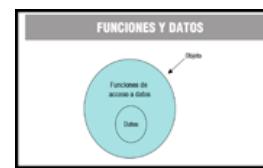
*John Johnson:* Primero resuelve el problema. Entonces, escribe el código.

## 2.3.- Características.

Cuando hablamos de Programación Orientada a Objetos, existen una serie de características que se deben cumplir. Cualquier lenguaje de programación orientado a objetos las debe contemplar. Las características más importantes del paradigma de la programación orientada a objetos son:

- ✓ **Abstracción.** Es el proceso por el cual definimos las características más importantes de un objeto, sin preocuparnos de cómo se escribirán en el código del programa, simplemente lo definimos de forma general. En la Programación Orientada a Objetos la herramienta más importante para soportar la abstracción es la **clase**. Básicamente, una clase es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Poder ver los objetos del mundo real que deseamos trasladar a nuestros programas, en términos abstractos, resulta de gran utilidad para un buen diseño del software, ya que nos ayuda a comprender mejor el problema y a tener una visión global de todo el conjunto. Por ejemplo, si pensamos en una clase **Vehículo** que agrupa las características comunes de todos ellos, a partir de dicha clase podríamos crear objetos como **Coche** y **Camión**. Entonces se dice que **Vehículo** es una abstracción de **Coche** y de **Camión**.
- ✓ **Modularidad.** Una vez que hemos representado el escenario del problema en nuestra aplicación, tenemos como resultado un conjunto de objetos software a utilizar. Este conjunto de objetos se crean a partir de una o varias clases. Cada clase se encuentra en un archivo diferente, por lo que la modularidad nos permite modificar las características de la clase que define un objeto, sin que esto afecte al resto de clases de la aplicación.
- ✓ **Encapsulación.** También llamada "**ocultamiento de la información**". La **encapsulación** o **encapsulamiento** es el mecanismo básico para ocultar la información de las partes internas de un objeto a los demás objetos de la aplicación. Con la encapsulación un objeto puede ocultar la información que contiene al mundo exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada. Por ejemplo, pensemos en un programa con dos objetos, un objeto **Persona** y otro **Coche**. **Persona** se comunica con el objeto **Coche** para llegar a su destino, utilizando para ello las acciones que **Coche** tenga definidas como por ejemplo conducir. Es decir, **Persona** utiliza **Coche** pero no sabe cómo funciona internamente, sólo sabe utilizar sus métodos o acciones.
- ✓ **Jerarquía.** Mediante esta propiedad podemos definir relaciones de jerarquías entre clases y objetos. Las dos jerarquías más importantes son la jerarquía "**es un**" llamada **generalización** o **especialización** y la jerarquía "**es parte de**", llamada **agregación**. Conviene detallar algunos aspectos:
  - ◆ La generalización o especialización, también conocida como **herencia**, permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases ya existentes (herencia múltiple). Por ejemplo, podemos crear la clase **CochedeCarreras** a partir de la clase **Coche**, y así sólo tendremos que definir las nuevas características que tenga.
  - ◆ La agregación, también conocida como **inclusión**, permite agrupar objetos relacionados entre sí dentro de una clase. Así, un **Coche** está formado por **Motor**, **Ruedas**, **Frenos** y **Ventanas**. Se dice que **Coche** es una agregación y **Motor**, **Ruedas**, **Frenos** Y **Ventanas** son agregados de **Coche**.
- ✓ **Polimorfismo.** Esta propiedad indica la capacidad de que varias clases creadas a partir de una antecesora realicen una misma acción de forma diferente. Por ejemplo, pensemos en la clase **Animal** y la acción de expresarse. Nos encontramos que cada tipo de **Animal** puede hacerlo de manera distinta, los **Perros** ladran, los **Gatos** maullan, las **Personas** hablamos, etc. Dicho de otra manera, el polimorfismo indica la posibilidad de tomar un objeto (de tipo **Animal**, por ejemplo), e indicarle que realice la acción de expresarse, esta acción será diferente según el tipo de mamífero del que se trate.

Los conceptos de herencia y polimorfismo serán trabajados en profundidad en las próximas unidades.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](https://creativecommons.org/licenses/by-nc/4.0/)

## 2.4.- Lenguajes de programación orientados a objetos.

Una panorámica de la evolución de los lenguajes de programación orientados a objetos hasta llegar a los utilizados actualmente es la siguiente:

- ✓ **Simula (1962).** El primer lenguaje con objetos fue B1000 en 1961, seguido por Sketchpad en 1962, el cual contenía clones o copias de objetos. Sin embargo, fue Simula el primer lenguaje que introdujo el concepto de clase, como elemento que incorpora datos y las operaciones sobre esos datos. En 1967 surgió Simula 67 que incorporaba un mayor número de tipos de datos, además del apoyo a objetos.
- ✓ **SmallTalk (1972).** Basado en Simula 67, la primera versión fue Smalltalk 72, a la que siguió Smalltalk 76, versión totalmente orientada a objetos. Se caracteriza por soportar las principales propiedades de la Programación Orientada a Objetos y por poseer un entorno que facilita el rápido desarrollo de aplicaciones. El Modelo-Vista-Controlador (MVC) fue una importante contribución de este lenguaje al mundo de la programación. El lenguaje Smalltalk ha influido sobre otros muchos lenguajes como C++ y Java.
- ✓ **C++ (1985).** C++ fue diseñado por Bjarne Stroustrup en los laboratorios donde trabajaba, entre 1982 y 1985. Lenguaje que deriva del C, al que añade una serie de mecanismos que le convierten en un lenguaje orientado a objetos. No tiene recolector de basura automática, lo que obliga a utilizar un destructor de objetos no utilizados. En este lenguaje es donde aparece el concepto de clase tal y como lo conocemos actualmente, como un conjunto de datos y funciones que los manipulan.
- ✓ **Eiffel (1986).** Creado en 1985 por Bertrand Meyer, recibe su nombre en honor a la famosa torre de París. Tiene una sintaxis similar a C. Soporta todas las propiedades fundamentales de los objetos, utilizado sobre todo en ambientes universitarios y de investigación. Entre sus características destaca la posibilidad de traducción de código Eiffel a Lenguaje C. Aunque es un lenguaje bastante potente, no logró la aceptación de C++ y Java.
- ✓ **Java (1995).** Diseñado por Gosling de Sun Microsystems a finales de 1995. Es un lenguaje orientado a objetos diseñado desde cero, que recibe muchas influencias de C++. Como sabemos, se caracteriza porque produce un bytecode que posteriormente es interpretado por la máquina virtual. La revolución de Internet ha influido mucho en el auge de Java.
- ✓ **C# (2000).** El lenguaje C#, también es conocido como Sharp. Fue creado por Microsoft, como una ampliación de C con orientación a objetos. Está basado en C++ y en Java. Una de sus principales ventajas que evita muchos de los problemas de diseño de C++.



Quasar (CC BY-SA)

En la actualidad muchos lenguajes de programación, incluso algunos que originalmente no fueron concebidos para ser orientados a objetos como Javascript o PHP, están migrando a la orientación a objetos.

### Autoevaluación

Relaciona los lenguajes de programación indicados con la característica correspondiente, escribiendo el número asociado a la característica en el hueco correspondiente.

#### Ejercicio de relacionar

Lenguaje de programación	Relación	Tiene la característica de que ...
Java	<input type="checkbox"/>	1. Fue el primer lenguaje que introdujo el concepto de clase.
SmallTalk	<input type="checkbox"/>	2. Introdujo el concepto del Modelo-Vista-Controlador.
Simula	<input type="checkbox"/>	3. Produce un bytecode para ser interpretado por la máquina virtual.
C++	<input type="checkbox"/>	4. Introduce el concepto de clase tal cual lo conocemos, con atributos y métodos.

Enviar

C++ fue diseñado basándose en C y añadiéndole la orientación a objetos. Posteriormente, surgiría Java, que recibió influencias de C++ y de Ada-95.

### 3.- Clases y Objetos. Características de los objetos.

#### Caso práctico



[Julius Schorzman \(CC0\)](#)

**María** ha hecho un descanso de cinco minutos. Se está tomando un café y está repasando los conceptos de Programación Orientada a Objetos. Piensa que este paradigma supone un cambio de enfoque con respecto a las técnicas tradicionales. Ahora lo que necesita es ahondar en el concepto de objeto, que parece ser el eje central de este modelo de programación.

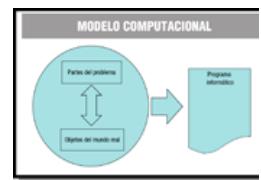
Al principio de la unidad veíamos que el mundo real está compuesto de objetos, y podemos considerar objetos casi cualquier cosa que podemos ver y sentir. Cuando escribimos un programa en un lenguaje orientado a objetos, debemos identificar cada una de las partes del problema con objetos presentes en el mundo real, para luego trasladarlos al modelo computacional que estamos creando.

En este contexto, un objeto de software es una representación de un objeto del mundo real, compuesto de una serie de características y un comportamiento específico. Pero ¿qué es más concretamente un objeto en Programación Orientada a Objetos? Veámoslo.

**Un objeto es un conjunto de datos con las operaciones definidas para ellos.** Los objetos tienen un **estado** y **un comportamiento**.

Por tanto, estudiando los objetos que están presentes en un problema podemos dar con la solución a dicho problema. Los objetos tienen unas características fundamentales que los distinguen:

- ✓ **Identidad.** Es la característica que permite diferenciar un objeto de otro. De esta manera, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Puede ser una dirección de memoria, el nombre del objeto o cualquier otro elemento que utilice el lenguaje para distinguirlos. Por ejemplo, dos vehículos que hayan salido de la misma cadena de fabricación y sean iguales aparentemente, son distintos porque tienen un código que los identifica.
- ✓ **Estado.** El estado de un objeto viene determinado por una serie de parámetros o atributos que lo describen, y los valores de éstos. Por ejemplo, si tenemos un objeto **Coche**, el estado estaría definido por atributos como **Marca**, **Modelo**, **Color**, **Cilindrada**, etc.
- ✓ **Comportamiento.** Son las acciones que se pueden realizar sobre el objeto. En otras palabras, son los métodos o procedimientos que realiza el objeto. Siguiendo con el ejemplo del objeto **Coche**, el comportamiento serían acciones como: **arrancar()**, **parar()**, **acelerar()**, **frenar()**, etc.



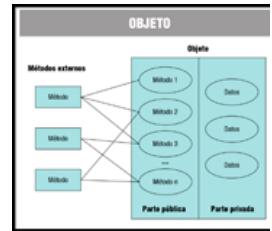
Ministerio de Educación (Elaboración propia)  
[\(CC BY-SA\)](#)

### 3.1.- Propiedades y métodos de los objetos.

Como acabamos de ver todo objeto tiene un estado y un comportamiento. Concretando un poco más, las partes de un objeto son:

- ✓ **Campos, Atributos o Propiedades:** Parte del objeto que almacena los datos. También se les denomina **Variables Miembro**. Estos datos pueden ser de cualquier tipo primitivo (`boolean`, `char`, `int`, `double`, etc) o ser su vez ser otro objeto. Por ejemplo, un objeto de la clase `Coche` puede tener un objeto de la clase `Ruedas`.
- ✓ **Métodos o Funciones Miembro:** Parte del objeto que lleva a cabo las operaciones sobre los atributos definidos para ese objeto.

La idea principal es que el objeto reúne en una sola entidad los datos y las operaciones, y para acceder a los datos privados del objeto debemos utilizar los métodos que hay definidos para ese objeto.



Ministerio de Educación (Elaboración propia)  
(CC BY-NC)

**La única forma de manipular la información del objeto es a través de sus métodos.** Es decir, si queremos saber el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo. De esta forma, evitamos que métodos externos puedan alterar los datos del objeto de manera inadecuada. **Se dice que los datos y los métodos están encapsulados dentro del objeto.**

## 3.2.- Interacción entre objetos.

Dentro de un programa los objetos se comunican llamando unos a otros a sus métodos. Los métodos están dentro de los objetos y describen el comportamiento de un objeto cuando recibe una llamada a uno de sus métodos. En otras palabras, cuando un objeto, `objeto1`, quiere actuar sobre otro, `objeto2`, tiene que ejecutar uno de sus métodos. Entonces se dice que el `objeto2` recibe un mensaje del `objeto1`.

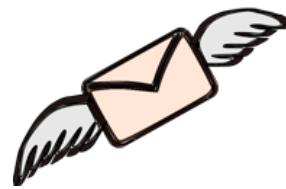
Un **mensaje** es la acción que realiza un objeto. Un **método** es la función o procedimiento al que se llama para actuar sobre un objeto.

Los distintos mensajes que puede recibir un objeto o a los que puede responder reciben el nombre de **protocolo** de ese objeto.

**El proceso de interacción entre objetos se suele resumir diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto determina "qué hacer con el mensaje" (ejecuta el código del método).** Cuando se ejecuta un programa se producen las siguientes acciones:

- ✓ Creación de los objetos a medida que se necesitan.
- ✓ Comunicación entre los objetos mediante el envío de mensajes unos a otros, o el usuario a los objetos.
- ✓ Eliminación de los objetos cuando no son necesarios para dejar espacio libre en la memoria del computador.

Srinath66 (CC BY)



Los objetos se pueden comunicar entre ellos invocando a los métodos de los otros objetos.

### Autoevaluación

Cuando un objeto, `objeto1`, ejecuta un método de otro, `objeto2`, se dice que el `objeto2` le ha mandado un mensaje al `objeto1`.

Verdadero  Falso

**Falso**

En este caso se dice que es el `objeto1` el que le ha mandado un mensaje al `objeto2`.

### 3.3.- Clases.

Hasta ahora hemos visto lo que son los objetos. Un programa informático se compone de muchos objetos, algunos de los cuales comparten la misma estructura y comportamiento. Si tuviéramos que definir su estructura y comportamiento objeto cada vez que queremos crear un objeto, estaríamos utilizando mucho código redundante. Por ello lo que se hace es crear una **clase**, que es una descripción de un conjunto de objetos que comparten una estructura y un comportamiento común. Y a partir de la clase, se crean tantas "copia" o "instancias" como necesitemos. Esas copias son los objetos de la clase. Realmente una clase es un "molde" que nos permite crear objetos de un determinado tipo.



Ministerio de Educación (Elaboración propia)  
(CC BY-NC)

Un ejemplo: definimos la clase **Vehículo**, que contendrá un conjunto de propiedades (**marca**, **potencia**, **velocidadMax**, etc) y un conjunto de métodos (aumentarVelocidad, ReducirVelocidad, etc). A partir de esa clase podremos instanciar objetos de tipo Vehículo: **objBMW318**, **objMercedes220**, etc. ¿Diferencia entre clase y objeto?. La clase no existe en memoria, realmente solo define la estructura de los objetos. Cuando un objeto es instanciado, se reserva un espacio de memoria para alojar sus datos. En realidad una clase se comporta como un tipo de datos mientras que un objeto es una instancia de ese tipo de datos, es decir, una variable de ese tipo de datos.

**Las clases constan de datos y métodos que resumen las características comunes de un conjunto de objetos.** Un programa informático está compuesto por un conjunto de clases, a partir de las cuales se crean objetos que interactúan entre sí.

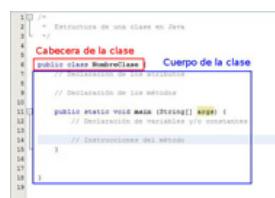
Si recuerdas, cuando utilizábamos los tipos de datos enumerados, los definíamos con la palabra reservada `enum` y la lista de valores entre llaves, y decíamos que un tipo de datos `enum` no es otra cosa que una especie de clase en Java. Efectivamente, todas las clases llevan su contenido entre llaves. Y una clase tiene la misma estructura que un tipo de dato enumerado, añadiéndole una serie de métodos y variables.

En otras palabras, **una clase es una plantilla o prototipo donde se especifican:**

- ✓ Los **atributos** comunes a todos los objetos de la clase.
- ✓ Los **métodos** que pueden utilizarse para manejar esos objetos.

Para declarar una clase en Java se utiliza la palabra reservada `class`. La declaración de una clase está compuesta por:

- ✓ **Cabecera de la clase.** La cabecera es un poco más compleja que como aquí definimos, pero por ahora sólo nos interesa saber que está compuesta por una serie de modificadores, en este caso hemos puesto `public` que indica que es una clase pública a la que pueden acceder otras clases del programa, la palabra reservada `class` y el nombre de la clase.
- ✓ **Cuerpo de la clase.** En él se especifican encerrados entre llaves los atributos y los métodos que va a tener la clase.



Ministerio de Educación (Elaboración propia)  
(CC BY-NC)

En la unidad anterior ya hemos utilizado clases, aunque aún no sabíamos su significado exacto. Por ejemplo, en los ejemplos de la unidad o en la tarea, estábamos utilizando clases, todas ellas eran clases principales, no tenían ningún atributo y el único método del que disponían era el método `main()`.

El método `main()` se utiliza para indicar que se trata de una clase principal, a partir de la cual va a empezar la ejecución del programa. Este método no aparece si la clase que estamos creando no va a ser la clase principal del programa.

## 4.- Utilización de objetos.

### Caso práctico

**María** sigue fuera de la oficina. Esta noche en casa quiere repasar conceptos sobre Programación Orientada a objetos, así que aprovecha un momento para llamar a **Juan** y le comenta:

-Ya sé todo sobre objetos -le dice- sólo que...

-¿Solo qué? -añade **Juan**.

-Solo me falta saber... ¿cómo se crea un objeto?

**Juan** sonríe ante la pregunta de **María**, y le explica que los objetos se crean como si fuera declarando una variable más, tan sólo que el tipo de datos de dicho objeto será una clase. Tras declararlos hay que instanciarlos con la orden `new` para reservar memoria para ellos, y después ya podremos utilizarlos, refiriéndonos a su contenido con el operador punto.

-Te mando un documento por email que lo explica todo muy bien.

-¡Ah, gracias! Esta noche le echo un vistazo -añade **María**.

Una vez que hemos creado una clase, podemos crear objetos en nuestro programa a partir de esas clases.

Cuando creamos un objeto a partir de una clase se dice que hemos creado una "**instancia de la clase**". A efectos prácticos, "objeto" e "instancia de clase" son términos similares. Es decir, nos referimos a objetos como instancias cuando queremos hacer hincapié que son de una clase particular.

Los **objetos** se crean a partir de las clases, y representan casos individuales de éstas.

Para entender mejor el concepto entre un objeto y su clase, piensa en un **molde de galletas** y **las galletas**. El molde sería la clase, que define las características del objeto, por ejemplo su forma y tamaño. Las galletas creadas a partir de ese molde son los objetos o instancias.

Otro ejemplo, imagina una clase `Persona` que reúna las características comunes de las personas (color de pelo, ojos, peso, altura, etc.) y las acciones que pueden realizar (crecer, dormir, comer, etc.). Posteriormente dentro del programa podremos crear un objeto `Trabajador` que esté basado en esa clase `Persona`. Entonces se dice que el objeto `Trabajador` es una instancia de la clase `Persona`, o que la clase `Persona` es una abstracción del objeto `Trabajador`.



Medea material (CC BY)

Cualquier objeto instanciado de una clase contiene una copia de todos los atributos definidos en la clase. En otras palabras, lo que estamos haciendo es reservar un espacio en la memoria del ordenador para guardar sus atributos y métodos. Por tanto, cada objeto tiene una **zona de almacenamiento propia** donde se guarda toda su información, que será distinta a la de cualquier otro objeto. A las variables miembro instanciadas también se les llama **variables instancia**. De igual forma, a los métodos que manipulan esas variables se les llama **métodos instancia**.

En el ejemplo del objeto `Trabajador`, las variables instancia serían `color_de_pelo`, `peso`, `altura`, etc. Y los métodos instancia serían `crecer()`, `dormir()`, `comer()`, etc.

### Autoevaluación

**Las variables instancia son un tipo de variables miembro.**

- Verdadero  Falso

**Verdadero**

Las variables miembro pueden ser variables instancia o variables de clase. Este último tipo es un caso

particular cuando el valor de la variable o atributo es compartido por todos los objetos de la clase.

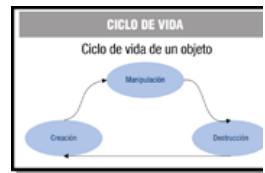
## 4.1.- Ciclo de vida de los objetos.

Todo programa en Java parte de una única clase, que como hemos comentado se trata de la clase principal. Esta clase ejecutará el contenido de su método **main()**, el cual será el que utilice las demás clases del programa, cree objetos y lance mensajes a otros objetos.

Las instancias u objetos tienen un tiempo de vida determinado. Cuando un objeto no se va a utilizar más en el programa, es destruido por el recolector de basura para liberar recursos que pueden ser reutilizados por otros objetos.

A la vista de lo anterior, podemos concluir que los objetos tienen un ciclo de vida, en el cual podemos distinguir las siguientes fases:

- ✓ **Creación**, donde se hace la reserva de memoria e inicialización de atributos.
- ✓ **Manipulación**, que se lleva a cabo cuando se hace uso de los atributos y métodos del objeto.
- ✓ **Destrucción**, eliminación del objeto y liberación de recursos.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](https://creativecommons.org/licenses/by-nc/4.0/)

## 4.2.- Declaración.

Para la creación de un objeto hay que seguir los siguientes pasos:

- ✓ **Declaración:** Definir el tipo de objeto.
- ✓ **Instanciación:** Creación del objeto utilizando el operador `new`.

Pero ¿en qué consisten estos pasos a nivel de programación en Java? Veamos primero cómo declarar un objeto. Para la definición del tipo de objeto debemos emplear la siguiente instrucción:

```
<tipo> nombre_objeto;
```

Donde:

- ✓ `tipo` es la clase a partir de la cual se va a crear el objeto, y
- ✓ `nombre_objeto` es el nombre de la variable referencia con la cual nos referiremos al objeto.

Los tipos referenciados o referencias se utilizan para guardar la dirección de los datos en la memoria del ordenador.

Nada más crear una referencia, ésta se encuentra vacía. Cuando una referencia a un objeto no contiene ninguna instancia se dice que es una referencia nula, es decir, que contiene el valor `null`. Esto quiere decir que la referencia está creada pero que el objeto no está instanciado todavía, por eso la referencia apunta a un objeto inexistente llamado "nulo".

Para entender mejor la declaración de objetos veamos un ejemplo. Cuando veíamos los tipos de datos en la Unidad 2, decíamos que Java proporciona un tipo de dato especial para los textos o cadenas de caracteres que era el tipo de dato `String`. Veíamos que realmente este tipo de dato es un **tipo referenciado** y creábamos una variable mensaje de ese tipo de dato de la siguiente forma:

```
String mensaje;
```

Los nombres de la clase empiezan con mayúscula, como `String`, y los nombres de los objetos con minúscula, como `mensaje`, así sabemos qué tipo de elemento utilizando.

Pues bien, `String` es realmente la clase a partir de la cual creamos nuestro objeto llamado `mensaje`.

Si observas poco se diferencia esta declaración de las declaraciones de variables que hacíamos para los tipos primitivos. Antes decíamos que `mensaje` era una variable del tipo de dato `String`. Ahora realmente vemos que `mensaje` es un objeto de la clase `String`. Pero `mensaje` aún no contiene el objeto porque no ha sido instanciado, veamos cómo hacerlo.

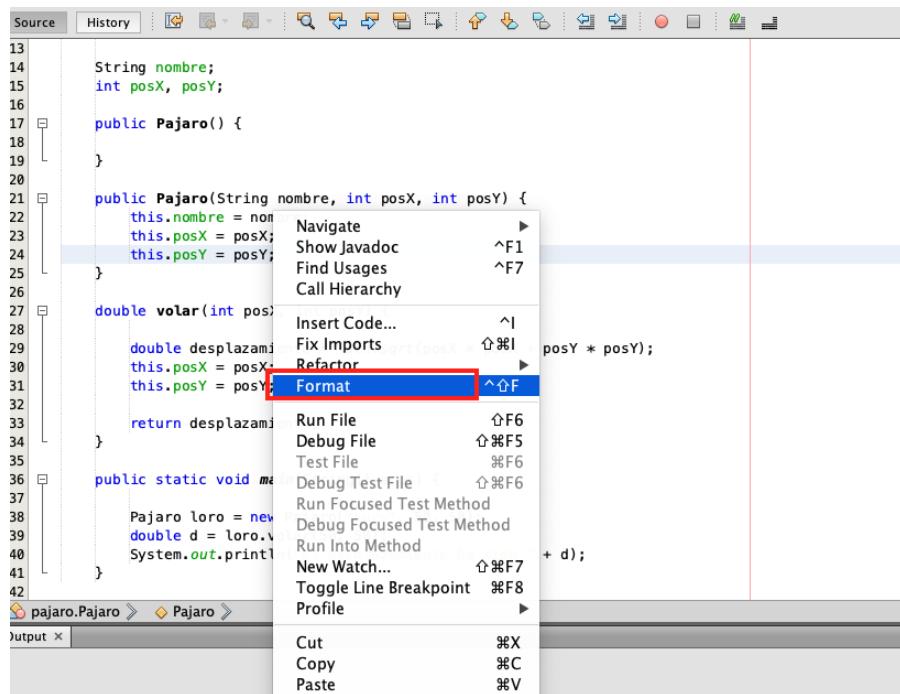
Por tanto, cuando creamos un objeto estamos haciendo uso de una variable que almacena la dirección de ese objeto en memoria. Esta variable es una **referencia** o un **tipo de datos referenciado**, porque no contiene el dato si no la posición del dato en la memoria del ordenador.

```
String saludo = new String ("Bienvenido a Java");  
String s; //s vale null  
s = saludo; //asignación de referencias
```

En las instrucciones anteriores, las variables `s` y `saludo` apuntan al mismo objeto de la clase `String`. Esto implica que cualquier modificación en el objeto `saludo` modifica también el objeto al que hace referencia la variable `s`, ya que realmente son el mismo.

**Para saber más**

Dentro de las ayudas que nos proporciona Netbeans, una muy interesante es el formateado de código. Como programadores debemos indentar el código a medida que lo escribimos si queremos mantener su legibilidad. Netbeans permite el formateado de código automático siguiente la estructura de bloques. Para ello solo tienes que seleccionar la opción que se muestra en la figura sobre el fichero de código fuente a formatear.



Ministerio de Educación y FP ([CC BY-NC](#))

## 4.3.- Instanciación.

Una vez creada la referencia al objeto, debemos crear la instancia u objeto que se va a guardar en esa referencia. Para ello utilizamos la orden **new** con la siguiente sintaxis:

```
nombre_objeto = new <Constructor_de_la_Clase>(<par1>, <par2>, ..., <parN>);
```

Donde:

- ✓ `nombre_objeto` es el nombre de la variable referencia con la cual nos referiremos al objeto,
- ✓ `new` es el operador para crear el objeto,
- ✓ `Constructor_de_la_Clase` es un método especial de la clase, que se llama igual que ella, y se encarga de inicializar el objeto, es decir, de dar unos valores iniciales a sus atributos, y
- ✓ `par1-parN`, son parámetros que puede o no necesitar el constructor para dar los valores iniciales a los atributos del objeto.

Durante la instanciación del objeto, se reserva memoria suficiente para el objeto. De esta tarea se encarga Java y juega un papel muy importante el **recolector de basura**, que se encarga de eliminar de la memoria los objetos no utilizados para que ésta pueda volver a ser utilizada.

De este modo, para instanciar un objeto `String`, haríamos lo siguiente:

```
mensaje = new String;
```

Así estaríamos instanciando el objeto `mensaje`. Para ello utilizaríamos el operador `new` y el constructor de la clase `String` a la que pertenece el objeto según la declaración que hemos hecho en el apartado anterior. A continuación utilizamos el constructor, que se llama igual que la clase, `String`.

En el ejemplo anterior el objeto se crearía con la cadena vacía (""), si queremos que tenga un contenido debemos utilizar parámetros en el constructor, así:

```
mensaje = new String ("El primer programa");
```

Java permite utilizar la clase `String` como si de un tipo de dato primitivo se tratara, por eso no hace falta utilizar el operador `new` para instanciar un objeto de la clase `String`.

La declaración e instanciación de un objeto puede realizarse en la misma instrucción, así:

```
String mensaje = new String ("El primer programa");
```

**¡Esta es la forma más útil de declarar un objeto e inicializarlo. Siempre que se puede inicializar el objeto, debemos utilizar esta forma de declararlo!**

### Recomendación

Crea un nuevo proyecto en Netbeans y añádele un clase principal con el método `main`. Es muy recomendable que vayas tecleando y probando cada uno de los ejemplos que se muestran en los contenidos. Si tienes errores de compilación corrígelos y después ejecuta la aplicación: **sin duda es la mejor y casi única forma de aprender**. Si tienes dudas puedes preguntar a tu profesor.

## 4.4.- Manipulación.

---

Una vez creado e instanciado el objeto ¿cómo accedemos a su contenido? Para acceder a los atributos y métodos del objeto utilizaremos el nombre del objeto seguido del **operador punto (.)** y el nombre del atributo o método que queremos utilizar.

Cuando utilizamos el operador punto se dice que estamos enviando un mensaje al objeto.

La forma general de enviar un mensaje a un objeto es:

```
nombre_objeto.mensaje
```

Por ejemplo, para acceder a las variables instancia o atributos se utiliza la siguiente sintaxis:

```
nombre_objeto.atributo
```

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis es:

```
nombre_objeto.método( [par1, par2, ..., parN] )
```

En la sentencia anterior `par1`, `par2`, etc. son los parámetros que utiliza el método. Aparece entre corchetes para indicar son opcionales.

Para entender mejor cómo se manipulan objetos vamos a utilizar un ejemplo. Para ello necesitamos la Biblioteca de Clases Java o API (Application Programming Interface - Interfaz de programación de aplicaciones). Uno de los paquetes de librerías o bibliotecas es `java.awt`. Este paquete contiene clases destinadas a la creación de objetos gráficos e imágenes. Vemos por ejemplo cómo crear un rectángulo.

En primer lugar instanciamos el objeto utilizando el método constructor, que se llama igual que el objeto, e indicando los parámetros correspondientes a la posición y a las dimensiones del rectángulo:

```
Rectangle rect = new Rectangle(50, 50, 150, 150);
```

Una vez instanciado el objeto rectángulo si queremos cambiar el valor de los atributos utilizamos el operador punto. Por ejemplo, para cambiar la dimensión del rectángulo:

```
rect.height=100;  
rect.width=100;
```

O bien podemos utilizar un método para hacer lo anterior:

```
rect.setSize(200, 200);
```

A continuación puedes acceder al código del ejemplo:

```

public class Manipular {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(50, 50, 250, 200);
        System.out.println("Las coordenadas x y y del rectángulo");
        System.out.println("x = " + rect.x + " y = " + rect.y);  

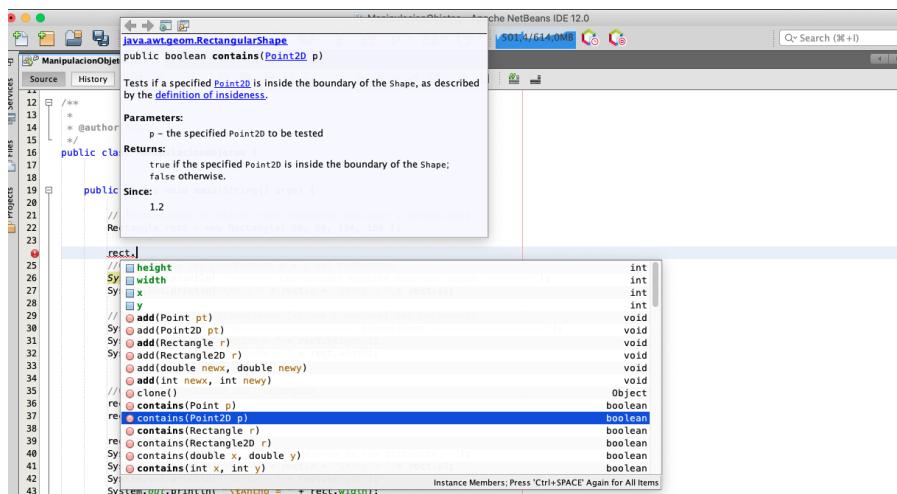
        System.out.println("Las dimensiones del rectángulo");
        System.out.println("width = " + rect.width + " height = " + rect.height);
        System.out.println("Las dimensiones totales del rectángulo");
        System.out.println("width + height = " + rect.width + rect.height);
    }
}

```

Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

## Debes conocer

Al hilo de las ayudas que nos proporciona Netbeans, a la hora de trabajar con objetos tenemos a nuestra disposición diversa funcionalidad que nos ayudará escribir código. Cuando instanciamos un objeto Java, comenzamos a utilizar su funcionalidad invocando sus métodos. A veces es complicado recordar los métodos disponibles en un objeto o incluso los parámetros que utiliza o la funcionalidad que implementa. Observa la siguiente imagen:



Ministerio de Educación y FP [\(CC BY-NC\)](#)

Si escribimos el identificador (nombre) de un objeto seguido del carácter ".", Netbeans nos muestra un listado con los métodos disponibles en ese objeto. Además, si seleccionamos con el cursor del ratón uno de ellos, nos muestra información del API estándar con la funcionalidad implementada y los parámetros utilizados. Si seleccionamos uno de ellos, Netbeans automáticamente añadirá la cabecera del método. **Nuestro trabajo será ajustar los parámetros.**

## Recomendación

Prueba el ejemplo anterior en Netbeans. Trata de teclear el código hasta que puedes ejecutarlo. Si tienes problemas, puedes descargar el proyecto [aquí](#).

## 4.5.- Destrucción de objetos y liberación de memoria.



pat00139 (CC BY)

Cuando un objeto deja de ser utilizado, es necesario liberar el espacio de memoria y otros recursos que poseía para poder ser reutilizados por el programa. A esta acción se le denomina **destrucción del objeto**.

En Java la destrucción de objetos corre a cargo del **recolector de basura (garbage collector)**. Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador `new`. Esto evita que los programadores tengan que preocuparse de realizar la liberación de memoria.

El recolector de basura se ejecuta en modo segundo plano y de manera muy eficiente para no afectar a la velocidad del programa que se está ejecutando. Lo que hace es que periódicamente va buscando objetos que ya no son referenciados, y cuando encuentra alguno lo marca para ser eliminado. Después los elimina en el momento que considera oportuno.

Justo antes de que un objeto sea eliminado por el recolector de basura, se ejecuta su método `finalize()`. Si queremos forzar que se ejecute el proceso de finalización de todos los objetos del programa podemos utilizar el método `runFinalization()` de la clase `System`. La clase `System` forma parte de la Biblioteca de Clases de Java y contiene diversas clases para la entrada y salida de información, acceso a variables de entorno del programa y otros métodos de diversa utilidad. Para forzar el proceso de finalización ejecutaríamos:

```
System.runFinalization();
```

### Autoevaluación

Las fases del ciclo de vida de un objeto son: Creación, Manipulación y Destrucción.

- Verdadero  Falso

**Verdadero**

Efectivamente, los objetos tienen un tiempo de vida limitado y esas son las fases que atraviesan durante su vida útil.

## 5.- Utilización de métodos.

### Caso práctico



Seudonimoanonomo (CC BY)

**María** está contenta con lo que está aprendiendo sobre Java, básicamente se trata de ampliar sus conocimientos y con la experiencia que ella tiene no le resultará difícil ponerse a programar en poco tiempo.

**Juan** ha comenzado el proyecto de la Clínica Veterinaria y quiere implicarse ella también lo antes posible. Además, están los dos becarios **Ana** y **Carlos**, que se han incorporado al proyecto hace poco y hay que comenzar a pensar en tareas para ellos.

Por lo pronto, **María** continúa con el documento facilitado por **Juan**, ahora tiene que ver cómo se utilizan los métodos, aunque intuye que no va a ser algo muy diferente a las funciones y procedimientos de cualquier otro lenguaje.



Mª Flor Moncada Añón (Elaboración propia)  
(CC BY-NC)

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, y que son ejecutadas cuando el método es invocado. Se definen en el cuerpo de la clase y posteriormente son instanciados para convertirse en **métodos instancia** de un objeto.

Para utilizar los métodos adecuadamente es conveniente conocer la estructura básica de que disponen.

Al igual que las clases, los métodos están compuestos por una **cabecera** y un **cuerpo**. La cabecera también tiene modificadores, en este caso hemos utilizado `public` para indicar que el método es público, lo cual quiere decir que le pueden enviar mensajes no sólo los métodos del objeto sino los métodos de cualquier otro objeto externo.

Dentro de un método nos encontramos el cuerpo del método que contiene el código de la acción a realizar. Las acciones que un método puede realizar son:

- ✓ **Iniciar** los atributos del objeto
- ✓ **Consultar** los valores de los atributos
- ✓ **Modificar** los valores de los atributos
- ✓ **Llamar** a otros métodos, del mismo del objeto o de objetos externos



Ministerio de Educación (Elaboración propia)  
(CC BY-NC)

### Citas para pensar

*Franklin Delano Roosevelt:* En cuestión de sentido común tomar un método y probarlo. Pero lo importante es probar algo.

## 5.1.- Parámetros y valores devueltos.

---

Los métodos se pueden utilizar tanto para consultar información sobre el objeto como para modificar su estado. La información consultada del objeto se devuelve a través de lo que se conoce como **valor de retorno**, y la modificación del estado del objeto, o sea, de sus atributos, se hace mediante la **lista de parámetros**.

En general, la lista de parámetros de un método se puede declarar de dos formas diferentes:

- ✓ **Por valor.** El valor de los parámetros no se devuelve al finalizar el método, es decir, cualquier modificación que se haga en los parámetros no tendrá efecto una vez se salga del método. Esto es así porque cuando se llama al método desde cualquier parte del programa, dicho método recibe una copia de los argumentos, por tanto cualquier modificación que haga será sobre la copia, no sobre las variables originales.
- ✓ **Por referencia.** La modificación en los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia lo que estamos haciendo es pasar la dirección del dato en memoria, por tanto cualquier cambio en el dato seguirá modificado una vez que salgamos del método.

En el lenguaje Java, todas las variables se pasan por valor, excepto los objetos que se pasan por referencia.

En Java, la declaración de un método tiene dos restricciones:

- ✓ **Un método siempre tiene que devolver un valor** (no hay valor por defecto). Este **valor de retorno** es el valor que devuelve el método cuando termina de ejecutarse, al método o programa que lo llamó. Puede ser un tipo primitivo, un tipo referenciado o bien el tipo void, que indica que el método no devuelve ningún valor.
- ✓ **Un método tiene un número fijo de argumentos.** Los argumentos son variables a través de las cuales se pasa información al método desde el lugar del que se llame, para que éste pueda utilizar dichos valores durante su ejecución. Los argumentos reciben el nombre de **parámetros** cuando aparecen en la declaración del método.

El **valor de retorno** es la información que devuelve un método tras su ejecución.

Según hemos visto en el apartado anterior, la cabecera de un método se declara como sigue:

```
public tipo_de_dato_devuelto nombre_metodo (lista_de_parametros);
```

Como vemos, el tipo de dato devuelto aparece después del modificador **public** y se corresponde con el valor de retorno.

La lista de parámetros aparece al final de la cabecera del método, justo después del nombre, encerrados entre signos de paréntesis y separados por comas. Se debe indicar el tipo de dato de cada parámetro así:

```
(tipo_parámetro1 nombre_parámetro1, ..., tipo_parámetroN nombre_parámetroN)
```

Cuando se llame al método, se deberá su nombre, seguido de los argumentos que deben coincidir con la lista de parámetros.

La **lista de argumentos** en la llamada a un método debe coincidir en número, tipo y orden con los **parámetros**, ya que de lo contrario se produciría un error de sintaxis.

## 5.2.- Constructores.

¿Recuerdas cuando hablábamos de la creación e instanciación de un objeto? Decíamos que utilizábamos el operador **new** seguido del nombre de la clase y una pareja de abrir-cerrar paréntesis. Además, el nombre de la clase era realmente el constructor de la misma, y lo definímos como un método especial que sirve para inicializar valores. En este apartado vamos a ver un poco más sobre los constructores.

**Un constructor es un método especial con el mismo nombre de la clase y que no devuelve ningún valor tras su ejecución.**

Cuando creamos un objeto debemos instanciarlo utilizando el constructor de la clase. Veamos la clase **Date** proporcionada por la Biblioteca de Clases de Java. Si queremos instanciar un objeto a partir de la clase **Date** tan sólo tendremos que utilizar el constructor seguido de una pareja de abrir-cerrar paréntesis:

```
Date fecha = new Date();
```



Con la anterior instrucción estamos creando un objeto **fecha** de tipo **Date**, que contendrá la fecha y hora actual del sistema.

Mª Flor Moncada Añón (Elaboración propia)  
[\(CC BY-NC\)](#)

La estructura de los constructores es similar a la de cualquier método, salvo que no tiene tipo de dato devuelto porque no devuelve ningún valor. Está formada por una cabecera y un cuerpo, que contiene la inicialización de atributos y resto de instrucciones del constructor.

El método constructor tiene las siguientes particularidades:

- ✓ **El constructor es invocado automáticamente en la creación de un objeto**, y sólo esa vez.
- ✓ **Los constructores no empiezan con minúscula**, como el resto de los métodos, ya que se llaman igual que la clase y las clases empiezan con letra mayúscula.
- ✓ **Puede haber varios constructores** para una clase.
- ✓ Como cualquier método, el constructor puede tener **parámetros** para definir qué valores dar a los atributos del objeto.
- ✓ El **constructor por defecto** es aquél que no tiene argumentos o parámetros. Cuando creamos un objeto llamando al nombre de la clase sin argumentos, estamos utilizando el constructor por defecto.
- ✓ **Es necesario que toda clase tenga al menos un constructor**. Si no definimos constructores para una clase, y sólo en ese caso, el compilador crea un constructor por defecto vacío, que inicializa los atributos a sus valores por defecto, según del tipo que sean: `0` para los tipos numéricos, `false` para los `boolean` y `null` para los tipo carácter y las referencias. Dicho constructor lo que hace es llamar al constructor sin argumentos de la superclase (clase de la cual hereda); si la superclase no tiene constructor sin argumentos se produce un error de compilación.

Cuando definimos constructores personalizados, el constructor por defecto deja de existir, y si no definimos nosotros un constructor sin argumentos cuando intentemos utilizar el constructor por defecto nos dará un error de compilación.

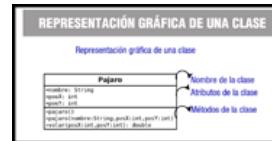
## 5.3.- El operador this.

Los constructores y métodos de un objeto suelen utilizar el operador `this`. Este operador sirve para referirse a los atributos de un objeto cuando estamos dentro de él. Sobre todo se utiliza cuando existe ambigüedad entre el nombre de un parámetro y el nombre de un atributo, entonces en lugar del nombre del atributo solamente escribiremos `this.nombre_atributo`, y así no habrá duda de a qué elemento nos estamos refiriendo.

Vamos a ilustrar mediante un ejemplo la utilización de objetos y métodos, así como el uso de parámetros y el operador `this`. Aunque la creación de clases la veremos en las siguientes unidades, en este ejercicio creamos una pequeña clase para que podamos instanciar el objeto con el que vamos a trabajar.

Las clases se suelen representar como un rectángulo, y dentro de él se sitúan los atributos y los métodos de dicha clase.

En la imagen, la clase `Pajaro` está compuesta por tres atributos, uno de ellos el nombre y otros dos que indican la posición del ave, `posX` y `posY`. Tiene dos métodos constructores y un método `volar()`. Como sabemos, los métodos constructores reciben el mismo nombre de la clase, y puede haber varios para una misma clase, dentro de ella se diferencian unos de otros por los parámetros que utilizan.



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

### Ejercicio resuelto

Dada una clase principal llamada `Pajaro`, se definen los atributos y métodos que aparecen en la imagen. Los métodos realizan las siguientes acciones:

- ✓ `pajaro()`. Constructor por defecto. En este caso, el constructor por defecto no contiene ninguna instrucción, ya que Java inicializa de forma automática las variables miembro, si no le damos ningún valor.
- ✓ `pajaro(String nombre, int posX, int posY)`. Constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.
- ✓ `volar(int posX, int posY)`. Método que recibe como argumentos dos enteros: `posX` y `posY`, y devuelve un valor de tipo double como resultado, usando la palabra clave `return`. El valor devuelto es el resultado de aplicar un desplazamiento de acuerdo con la siguiente fórmula:  
$$\text{desplazamiento} = \sqrt{\text{posX} \cdot \text{posX} + \text{posY} \cdot \text{posY}}$$

Diseña un programa que utilice la clase `Pajaro`, cree una instancia de dicha clase y ejecute sus métodos.

[Mostrar retroalimentación](#)

Lo primero que debemos hacer es crear la clase `Pajaro`, con sus métodos y atributos. De acuerdo con los datos que tenemos, el código de la clase sería el siguiente:

```
public class Pajaro {  
    String nombre;  
    int posX, posY;  
    public Pajaro() {}  
    public Pajaro(String nombre, int posX, int posY) {  
        this.nombre = nombre;  
        this.posX = posX;  
        this.posY = posY;  
    }  
    double volar(int posX, int posY) {  
        double desplazamiento = Math.sqrt(posX * posX + posY * posY);  
        this.posX = posX;  
        this.posY = posY;  
        return desplazamiento;  
    }  
}
```

Debemos tener en cuenta que se trata de una clase principal, lo cual quiere decir que debe contener un método `main()` dentro de ella. En el método `main()` vamos a situar el código de nuestro programa. El ejercicio dice que tenemos que crear una instancia de la clase y ejecutar sus métodos, entre los que están el constructor y el método `volar()`. También es conveniente imprimir el resultado de ejecutar el método `volar()`. Por tanto, lo que haría el programa sería:

- ✓ **Crear** un objeto de la clase e inicializarlo.
- ✓ **Invocar** al método `volar`.
- ✓ **Imprimir** por pantalla la distancia recorrida.

Para inicializar el objeto utilizaremos el constructor con parámetros, después ejecutaremos el método `volar()` del objeto creado y finalmente imprimiremos el valor que nos devuelve el método. El código de la clase `main()` quedaría como sigue:

```
public static void main(String[] args) {  
    Pajaro toro = new Pajaro("Lucy", 59.58);  
    double d = toro.volante();  
    System.out.println("El desplazamiento ha sido " + d);  
}
```

Si ejecutamos nuestro programa el resultado sería el siguiente:



Si quieres acceder al archivo completo del ejercicio puedes utilizar el siguiente enlace:

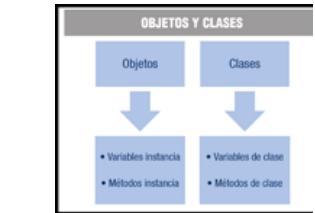
[Ejercicio resuelto clase Pajaro](#)

## 5.4.- Métodos estáticos.

Cuando trabajábamos con cadenas de caracteres utilizando la clase `String`, veíamos las operaciones que podíamos hacer con ellas: obtener longitud, comparar dos cadenas de caracteres, cambiar a mayúsculas o minúsculas, etc. Pues bien, sin saberlo estabamos utilizando métodos estáticos definidos por Java para la clase `String`. Pero ¿qué son los métodos estáticos? Veámoslo.

Los **métodos estáticos** son aquellos métodos definidos para una clase que se pueden usar directamente, sin necesidad de crear un objeto de dicha clase. También se llaman **métodos de clase**.

Para llamar a un método estático utilizaremos:



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

- ✓ El nombre del método, si lo llamamos desde la misma clase en la que se encuentra definido.
- ✓ El nombre de la clase, seguido por el operador punto (.) más el nombre del método estático, si lo llamamos desde una clase distinta a la que se encuentra definido:

`nombre_clase.nombre_metodo_estatico`

- ✓ El nombre del objeto, seguido por el operador punto (.) más el nombre del método estático. Utilizaremos esta forma cuando tengamos un objeto instanciado de la clase en la que se encuentra definido el método estático, y no podamos utilizar la anterior:

`nombre_objeto.nombre_metodo_estatico`

Los métodos estáticos no afectan al estado de los objetos instanciados de la clase (variables instancia), y **suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase**. Por ejemplo, si necesitamos contar el número de objetos instanciados de una clase, podríamos utilizar un método estático que fuera incrementando el valor de una variable entera de la clase conforme se van creando los objetos.

En la Biblioteca de Clases de Java existen muchas clases que contienen métodos estáticos. Pensemos en las clases que ya hemos utilizado en unidades anteriores, como hemos comentado la clase `String` con todas las operaciones que podíamos hacer con ella y con los objetos instanciados a partir de ella. O bien la clase `Math`, para la conversión de tipos de datos. Todos ellos son métodos estáticos que la API de Java define para esas clases. Lo importante es que tengamos en cuenta que al tratarse de métodos estáticos, para utilizarlos no necesitamos crear un objeto de dichas clases.

### Autoevaluación

**Los métodos estáticos, también llamados métodos instancia, suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase.**

Verdadero  Falso

**Falso**

Los métodos estáticos no pueden ser métodos instancia porque se utilizan directamente, sin crear un objeto de la clase.

## 6.- Librerías de objetos (paquetes).

### Caso práctico



jmerlelo (CC BY-SA)

-¡Vaya! -exclama **María**- No consigo encontrar la clase `Persona` dentro del conjunto de clases que hasta ahora he creado.

-¿Por qué no pruebas a dividir las clases en paquetes? -pregunta **Juan**- En un paquete agrupa las que estén relacionadas, y así te será más fácil encontrar una clase la próxima vez. Además puedes crear paquetes dentro de otros, como si fuera una estructura de directorios.

-¿Ah sí? Pues es justo lo que necesito para resolver este desorden. Voy a ponerme con ello -añade **María**.

Conforme nuestros programas se van haciendo más grandes, el número de clases va creciendo. Meter todas las clases en único directorio no ayuda a que estén bien organizadas, lo mejor es hacer **grupos de clases**, de forma que todas las clases que estén relacionadas o traten sobre un mismo tema estén en el mismo grupo.

Un **paquete** de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan de un tema común.

**Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete. Es por ello por lo que se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.**

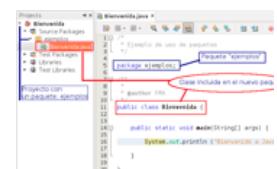
Java nos ayuda a organizar las clases en **paquetes**. En cada fichero .java que hagamos, al principio, podemos indicar a qué **paquete** pertenece la clase que hagamos en ese fichero.

Los paquetes se declaran utilizando la palabra clave `package` seguida del nombre del paquete. Para establecer el paquete al que pertenece una clase hay que poner una sentencia de declaración como la siguiente al principio de la clase:

```
package Nombre_de_Paquete;
```

Por ejemplo, si decidimos agrupar en un paquete "ejemplos" un programa llamado "Bienvenida", pondríamos en nuestro fichero `Bienvenida.java` lo siguiente:

El código es exactamente igual que como hemos venido haciendo hasta ahora, solamente hemos añadido la línea "`package ejemplos;`" al principio. En la imagen se muestra cómo aparecen los paquetes en el entorno integrado de Netbeans.



Mª Flor Moncada Arón (Elaboración propia)  
(CC BY-NC)

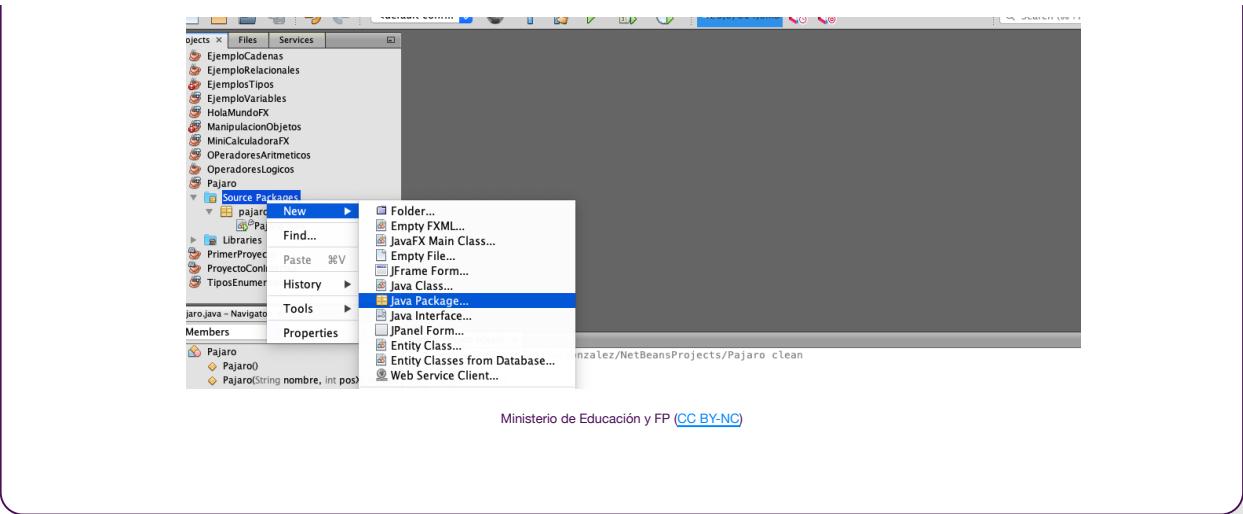
### Debes conocer

Para crear paquetes en Netbeans, tan solo tienes que hacer click con el botón derecho sobre:

- El paquete raíz **Source Packages**.
- En un paquete/subpaquete ya existente.

y seleccionar la opción **New - Java Package**.

Puedes verlo en la siguiente imagen.



## 6.1.- Sentencia import.

Cuando queremos utilizar una clase que está en un paquete distinto a la clase que estamos utilizando, se suele utilizar la sentencia `import`. Por ejemplo, si queremos utilizar la clase `Scanner` que está en el paquete `java.util` de la Biblioteca de Clases de Java, tendremos que utilizar esta sentencia:

```
import java.util.Scanner;
```

Se pueden importar todas las clases de un paquete, así:

```
import java.awt.*;
```

Esta sentencia debe aparecer al principio de la clase, justo después de la sentencia `package`, si ésta existiese.

También podemos utilizar la clase sin sentencia `import`, en cuyo caso cada vez que queramos usarla debemos indicar su ruta completa:

```
java.util.Scanner teclado = new java.util.Scanner (System.in);
```

Hasta aquí todo correcto. Sin embargo, al trabajar con paquetes, Java nos obliga a organizar los directorios, compilar y ejecutar de cierta forma para que todo funcione adecuadamente.

### Para saber más

Te proponemos el siguiente enlace para repasar conceptos que hemos visto a lo largo de la unidad sobre programación orientada a objetos y utilización de clases:

**Tutorial de Java en español- Capítulo 13 – Clases y Objetos.**

<https://www.youtube.com/embed/lhtCbil8JuE>

Resumen textual alternativo

La siguiente parte del vídeo habla sobre la utilización de objetos y clases en Java:

**Tutorial de Java en español- Capítulo 14 – Clases y Objetos.**

<https://www.youtube.com/embed/k92WaQyzVd4>

Resumen textual alternativo

## 6.2.- Compilar y ejecutar clases con paquetes.

Si hacemos que `Bienvenida.java` pertenezca al paquete `ejemplos`, debemos crear un subdirectorio "ejemplos" y meter dentro el archivo `Bienvenida.java`.

Por ejemplo, en Linux tendríamos esta estructura de directorios:

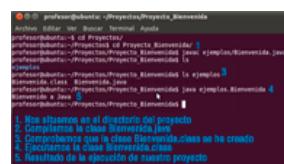
```
/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/Bienvenida.java
```

Debemos tener cuidado con las mayúsculas y las minúsculas, para evitar problemas, tenemos que poner el nombre en "package" exactamente igual que el nombre del subdirectorio.

Para **compilar** la clase `Bienvenida.java` que está en el paquete `ejemplos` debemos situarnos en el directorio padre del paquete y compilar desde ahí:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida  
$ javac ejemplos/Bienvenida.java
```

Si todo va bien, en el directorio `ejemplos` nos aparecerá la clase compilada `Bienvenida.class`.



M<sup>a</sup> Flor Moncada Añón (Elaboración propia)  
[\(GNU/GPL\)](#)

Para ejecutar la clase compilada `Bienvenida.class` que está en el directorio `ejemplos`, debemos seguir situados en el directorio padre del paquete. El nombre completo de la clase es "paquete.clase", es decir "`ejemplos.Bienvenida`". Los pasos serían los siguientes:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida  
$ java ejemplos/Bienvenida  
Bienvenido a Java
```

Si todo es correcto, debe salir el mensaje "`Bienvenido a Java`" por la pantalla.

## 6.3.- Jerarquía de paquetes.

Para organizar mejor las cosas, un **paquete**, en vez de clases, también puede contener otros paquetes. Es decir, podemos hacer **subpaquetes** de los paquetes y subpaquetes de los subpaquetes y así sucesivamente. Esto permite agrupar paquetes relacionados en un paquete más grande. Por ejemplo, si quiero dividir mis clases de ejemplos en ejemplos básicos y ejemplos avanzados, puedo poner más niveles de paquetes separando por puntos:

```
package ejemplos.basicos;  
  
package ejemplos.avanzados;
```



[odd.note \(CC BY\)](#)

A nivel de sistema operativo, tendríamos que crear los subdirectorios `basicos` y `avanzados` dentro del directorio `ejemplos`, y meter ahí las clases que correspondan.

Para compilar, en el directorio del proyecto habría que compilar poniendo todo el path hasta llegar a la clase. Es decir, el nombre de la clase va con todos los paquetes separados por puntos, por ejemplo `ejemplos.basicos.Bienvenida`.

La estructura de directorios en el sistema operativo cuando usamos subpaquetes sería:

```
/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/basicos/HolaMundo.java
```

Y la compilación y ejecución sería:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida  
$ javac ejemplos/basicos/Bienvenida.java  
$ java ejemplos/basicos/Bienvenida  
Hola Mundo
```



Mª Flor Moncada Añón (Elaboración propia)  
[\(GNU/GPL\)](#)

La Biblioteca de Clases de Java se organiza haciendo uso de esta jerarquía de paquetes. Así por ejemplo, si quiero acceder a la clase `Date`, tendré que importarla indicando su ruta completa, o sea, `java.util.Date`. Este tipo de instrucciones son añadidas de forma automática cuando utilizamos un IDE.

```
import java.util.Date;
```

### Citas para pensar

Tan bueno como es heredar una biblioteca, es mejor colecciónar una.

**Augustine Birrel.**

## 6.4.- Librerías Java.

Cuando descargamos el entorno de compilación y ejecución de Java, obtenemos la API de Java. Como ya sabemos, se trata de un conjunto de bibliotecas que nos proporciona paquetes de clases útiles para nuestros programas.

Utilizar las clases y métodos de la Biblioteca de Java nos va ayudar a reducir el tiempo de desarrollo considerablemente, por lo que es importante que aprendamos a consultarla y conozcamos las clases más utilizadas.

Los paquetes más importantes que ofrece el lenguaje Java son:

- ✓ **java.io.** Contiene las clases que gestionan la entrada y salida, ya sea para manipular ficheros, leer o escribir en pantalla, en memoria, etc. Este paquete contiene por ejemplo la clase `BufferedReader` que se utiliza para la entrada por teclado.
- ✓ **java.lang.** Contiene las clases básicas del lenguaje. Este paquete no es necesario importarlo, ya que es importado automáticamente por el entorno de ejecución. En este paquete se encuentra la clase `Object`, que sirve como raíz para la jerarquía de clases de Java, o la clase `System` que ya hemos utilizado en algunos ejemplos y que representa al sistema en el que se está ejecutando la aplicación. También podemos encontrar en este paquete las clases que "envuelven" los tipos primitivos de datos. Lo que proporciona una serie de métodos para cada tipo de dato de utilidad, como por ejemplo las conversiones de datos.
- ✓ **java.util.** Biblioteca de clases de utilidad general para el programador. Este paquete contiene por ejemplo la clase `Scanner` utilizada para la entrada por teclado de diferentes tipos de datos, la clase `Date`, para el tratamiento de fechas, etc.
- ✓ **java.math.** Contiene herramientas para manipulaciones matemáticas.
- ✓ **java.awt.** Incluye las clases relacionadas con la construcción de interfaces de usuario, es decir, las que nos permiten construir ventanas, cajas de texto, botones, etc. Algunas de las clases que podemos encontrar en este paquete son `Button`, `TextField`, `Frame`, `Label`, etc.
- ✓ **java.swing.** Contiene otro conjunto de clases para la construcción de interfaces avanzadas de usuario. Los componentes que se engloban dentro de este paquete se denominan componentes Swing, y suponen una alternativa mucho más potente que `AWT` para construir interfaces de usuario.
- ✓ **java.net.** Conjunto de clases para la programación en la red local e Internet.
- ✓ **java.sql.** Contiene las clases necesarias para programar en Java el acceso a las bases de datos.
- ✓ **java.security.** Biblioteca de clases para implementar mecanismos de seguridad.

Como se puede comprobar Java ofrece una completa jerarquía de clases organizadas a través de paquetes.

### Para saber más

En el siguiente enlace puedes acceder a la información oficial sobre la Biblioteca de Clases de Java (está en Inglés).

[Información oficial sobre la Biblioteca de Clases de Java.](#)

En el siguiente vídeo se explica de manera interesante el paquete básico `java.lang`:

[Clases que componen el paquete java.lang.](#)

<https://www.youtube.com/embed/bACxc1RAed4>

[Resumen textual alternativo](#)

## 7.- Programación de la consola: entrada y salida de la información.

### Caso práctico

**Juan** va a realizar algunas pruebas con el proyecto de la Clínica Veterinaria. Le comenta a **María** que lo que tienen ahora mismo es la estructura básica del proyecto, que básicamente se trata de la definición de algunas clases y objetos que se van a utilizar. Van a necesitar introducir datos por pantalla para ver cómo se comportan esos objetos, y mostrar por pantalla el resultado de manipularlos. Para ello utilizarán las clases `System` y `Scanner`.

—Estas clases ya las hemos utilizado anteriormente, están en los paquetes `java.lang` y `java.util`, respectivamente, observa en el siguiente código cómo las utilizo —dice **Juan**.

—De acuerdo, enséñame ese código —comenta **María**.

Los programas a veces necesitan acceder a los recursos del sistema, como por ejemplo los dispositivos de entrada/salida estándar, para recoger datos de teclado o mostrar datos por pantalla.

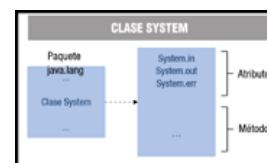
En Java, la entrada por teclado y la salida de información por pantalla se hace mediante la clase `System` del paquete `java.lang` de la Biblioteca de Clases de Java.

Como cualquier otra clase, está compuesta de métodos y atributos. Los atributos de la clase `System` son tres objetos que se utilizan para la entrada y salida estándar. Estos objetos son los siguientes:

- ✓ `System.in`. Entrada estándar: teclado.
- ✓ `System.out`. Salida estándar: pantalla.
- ✓ `System.err`. Salida de error estándar, se produce también por pantalla, pero se implementa como un fichero distinto al anterior para distinguir la salida normal del programa de los mensajes de error. Se utiliza para mostrar mensajes de error.

No se pueden crear objetos a partir de la clase `System`, sino que se utiliza directamente llamando a cualquiera de sus métodos con el operador de manipulación de objetos, es decir, el operador punto ():

```
System.out.println("Bienvenido a Java");
```



Ministerio de Educación (Elaboración propia)  
[\(CC BY-NC\)](#)

### Para saber más

En el siguiente enlace puedes consultar los atributos y métodos de la clase `System` del paquete `java.lang` perteneciente a la Biblioteca de Clases de Java:

[Clase System de Java.](#)

### Autoevaluación

**Texto de la pregunta tipo verdadero o falso:**

La clase `System` del paquete `java.io`, como cualquier clase, está formada por métodos y atributos, y además es

una clase que no se puede instanciar, sino que se utiliza directamente.

- Verdadero  Falso

**Falso**

La clase `System` pertenece al paquete `java.lang`.

## 7.1.- Conceptos sobre la clase System.

La lectura por teclado es muy importante cuando empezamos a hacer nuestros primeros programas. Para entender mejor en qué consiste la clase `System`, y en particular el objeto `System.in` vamos a describirlo más detenidamente.

En el apartado anterior hemos dicho que `System.in` es un atributo de la clase `System`, que está dentro del paquete `java.lang`. Pero además, si consultamos la Biblioteca de Clases de Java, nos damos cuenta que es un objeto, y como todos los objetos debe ser instanciado. En efecto, volviendo a consultar la biblioteca de clases nos damos cuenta que `System.in` es una instancia de una clase de java que se llama `InputStream`.

Field Summary		
Fields	Modifier and Type	Field and Description
	static <code>PrintStream</code>	<code>err</code> The "standard" error output stream
	static <code>InputStream</code>	<code>in</code> The "standard" input stream.
	static <code>PrintStream</code>	<code>out</code> The "standard" output stream.

Oracle . Captura de pantalla de la web de Oracle  
(Todos los derechos reservados)

En Java, `InputStream` nos permite leer en bytes, desde teclado, un archivo o cualquier otro dispositivo de entrada. Con esta clase podemos utilizar por ejemplo el método `read()` que permite leer un byte de la entrada o `skip(long n)`, que salta `n` bytes de la entrada. Pero lo que realmente nos interesa es poder leer texto o números, no bytes, para hacernos más cómoda la entrada de datos. Para ello se utilizan las clases:

- ✓ `InputStreamReader`. Convierte los bytes leídos en caracteres. Particularmente, nos va a servir para convertir el objeto `System.in` en otro tipo de objeto que nos permita leer caracteres.
- ✓ `BufferedReader`. Lee hasta un fin de línea. Esta es la clase que nos interesa utilizar, pues tiene un método `readLine()` que nos va a permitir leer caracteres hasta el final de línea.

La forma de instanciar estas clases para usarlas con `System.in` es la siguiente:

```
InputStreamReader isr = new InputStreamReader(System.in);  
  
BufferedReader br = new BufferedReader (isr);
```



Ministerio de Educación ([CC BY-NC](http://creativecommons.org/licenses/by-nc/3.0/))

En el código anterior hemos creado un `InputStreamReader` a partir de `System.in` y pasamos dicho `InputStreamReader` al constructor de `BufferedReader`. El resultado es que las lecturas que hagamos con el objeto `br` son en realidad realizadas sobre `System.in`, pero con la ventaja de que podemos leer una línea completa. Así, por ejemplo, si escribimos una A, con:

```
String cadena = br.readLine();
```

Obtendremos en `cadena` una "A".

Sin embargo, seguimos necesitando hacer la conversión si queremos leer números. Por ejemplo, si escribimos un entero 32, en cadena obtendremos "32". Si recordamos, para convertir cadenas de texto a enteros se utiliza el método estático `parseInt()` de la clase `Integer`, con lo cual la lectura la haríamos así:

```
int numero = Integer.parseInt (br.readLine());
```

## 7.2.- Entrada por teclado. Clase System.

A continuación vamos a ver un ejemplo de cómo utilizar la clase `System` para la entrada de datos por teclado en Java.

Como ya hemos visto en unidades anteriores, para compilar y ejecutar el ejemplo puedes utilizar las órdenes `javac` y `java`, o bien crear un nuevo proyecto en Netbeans y copiar el código que se proporciona en el archivo anterior.



Stockbyte CD-DVD Num. CDv43 ([CC BY-NC](#))

M<sup>a</sup> Flor Moncada Añón ([CC BY-NC](#))

### Código de entrada por teclado con la clase System.

Observa que hemos metido el código entre excepciones `try-catch`. Cuando en nuestro programa falla algo, por ejemplo la conversión de un `String` a `int`, Java nos avisa lanzando excepciones. Si "capturamos" esa excepción en nuestro programa, podemos avisar al usuario de qué ha pasado. Esto es conveniente porque si no tratamos la `System` excepción seguramente el programa se pare y no siga ejecutándose. El control de excepciones lo veremos en unidades posteriores, ahora sólo nos basta saber que en las llaves del `try` colocamos el código que puede fallar y en las llaves del `catch` el tratamiento de la excepción.

## 7.3.- Entrada por teclado. Clase Scanner.

La entrada por teclado que hemos visto en el apartado anterior tiene el inconveniente de que sólo podemos leer de manera fácil tipos de datos `String`. Si queremos leer otros tipos de datos deberemos convertir la cadena de texto leída en esos tipos de datos.

El kit de Desarrollo de Java, a partir de su versión 1.5, incorpora la clase `java.util.Scanner`, la cual permite leer tipos de datos `String`, `int`, `long`, etc., a través de la consola de la aplicación. Por ejemplo para leer un tipo de datos entero por teclado sería:

```
Scanner teclado = new Scanner (System.in);

int i = teclado.nextInt();
```

O bien esta otra instrucción para leer una línea completa, incluido texto, números o lo que sea:

```
String cadena = teclado.nextLine();
```

En las instrucciones anteriores hemos creado un objeto de la clase `Scanner` llamado `teclado` utilizando el constructor de la clase, al cual le hemos pasado como parámetro la entrada básica del sistema `System.in` que por defecto está asociada al teclado.

Para conocer cómo funciona un objeto de la clase `Scanner` te proporcionamos el siguiente ejemplo:

[Código de entrada por teclado con la clase Scanner.](#)



Oracle. Captura de pantalla de la web de Oracle (Todos los derechos reservados)



Mª Flor Moncada Añón ([CC BY-NC](#))

### Para saber más

Si quieres conocer algo más sobre la clase `Scanner` puedes consultar el siguiente enlace:

[Capturar datos desde teclado con Scanner.](#)

### Debes conocer

Al instanciar un objeto de una determinada clase, es probable que previamente tengamos que importarla: en otro caso obtendremos un error de compilación pues Netbeans no conoce el paquete que contiene la clase a instanciar.

Como se puede observar en la imagen, si haces click sobre el círculo rojo en el número de línea con el error, Netbeans nos sugiere soluciones al error. Fíjate que la primera sugerencia es importar la clase `Scanner`. Se seleccionar, Netbeans automáticamente generará el `import` y el error estará solucionado. ¡Mas ayudas para el programador!

A screenshot of an IDE interface showing a Java code editor. The code is as follows:

```
lacionales
pos
riables
ladoScanner
Packages
adatecladoscanner
EntradaTecladoScanner.java
ckages
s
rables
FX
onObjetos
doraFX
sAritmeticos
sLogicos
ecto
t <empty>
ladoScanner
TecladoScanner()
ring[] args)
    package entradatecladoscanner;
    /*
     * @author FMA
     */
    public class EntradaTecladoScanner {
        public static void main(String[] args) {
            // Creamos objeto teclado
            Scanner teclado = new Scanner(System.in);
            float salario;
            // Entrada de datos
            System.out.println("Nombre: ");
            nombre=teclado.nextLine();
            System.out.println("Edad: ");
            edad=teclado.nextInt();
            System.out.println("Estudios: ");
            ...
        }
    }

```

The cursor is at the end of the line 'Scanner teclado = new Scanner(System.in);'. A tooltip box is open, listing several suggestions related to the 'Scanner' class:

- Add import for java.util.Scanner
- Create class "Scanner" with constructor "Scanner(java.io.InputStream)" in package entradatecladoscanner (Source Packages)
- Create class "Scanner" with constructor "Scanner()" in package entradatecladoscanner (Source Packages)
- Create class "Scanner" in entradatecladoscanner.EntradaTecladoScanner
- Create class "Scanner" in entradatecladoscanner.EntradaTecladoScanner

Ministerio de Educación y FP [\(CC BY-NC\)](#)

## 7.4.- Salida por pantalla.

La salida por pantalla en Java se hace con el objeto `System.out`. Este objeto es una instancia de la clase `PrintStream` del paquete `java.lang`. Si miramos la API de `PrintStream` obtendremos la variedad de métodos para mostrar datos por pantalla, algunos de estos son:

- ✓ `void print(String s)`: Escribe una cadena de texto.
- ✓ `void println(String x)`: Escribe una cadena de texto y termina la línea.
- ✓ `void printf(String format, Object... args)`: Escribe una cadena de texto utilizando formato.

En la orden `print` y `println`, cuando queremos escribir un mensaje y el valor de una variable debemos utilizar el operador de concatenación de cadenas (+), por ejemplo:

The screenshot shows the Java API documentation for the `PrintStream` class. It includes the package name `java.io`, the class name `PrintStream`, and its inheritance path: `java.lang.Object` → `java.io.OutputStream` → `java.io.FilterOutputStream` → `java.io.PrintStream`. It also lists the `All Implemented Interfaces` as `Closeable, Flushable, Appendable, AutoCloseable` and the `Direct Known Subclasses` as `LogStream`. A copyright notice at the bottom states: "Oracle. Captura de pantalla de la web de Oracle. (Todos los derechos reservados)".

```
System.out.println("Bienvenido, " + nombre);
```

Escribe el mensaje de "Bienvenido, Carlos", si el valor de la variable `nombre` es Carlos.

Las órdenes `print` y `println` todas las variables que escriben las consideran como cadenas de texto sin formato, por ejemplo, no sería posible indicar que escriba un número decimal con dos cifras decimales o redondear las cifras, o escribir los puntos de los miles, por ejemplo. Para ello se utiliza la orden `printf()`.

La orden `printf()` utiliza unos códigos de conversión para indicar si el contenido a mostrar de qué tipo es. Estos códigos se caracterizan porque llevan delante el símbolo %, algunos de ellos son:

- ✓ `%c`: Escribe un carácter.
- ✓ `%s`: Escribe una cadena de texto.
- ✓ `%d`: Escribe un entero.
- ✓ `%f`: Escribe un número en punto flotante.
- ✓ `%e`: Escribe un número en punto flotante en notación científica.

Por ejemplo, si queremos escribir el número `float` 12345.1684 con el punto de los miles y sólo dos cifras decimales la orden sería:

```
System.out.printf("%,.2f\n", 12345.1684);
```

Esta orden mostraría el número `12.345,17` por pantalla.

Estas órdenes pueden utilizar las secuencias de escape que vimos en unidades anteriores, como "`\n`" para crear un salto de línea, "`\t`" para introducir un salto de tabulación en el texto, etc.

### Para saber más

Si quieres conocer algo más sobre la orden `printf()` en el siguiente enlace tienes varios ejemplos de utilización:

[Salida de datos con la orden `printf\(\)`.](#)

En este otro enlace puedes ver un ejemplo interesante de uso de la clase `Scanner`:

[Ejemplo de entrada y salida de datos.](#)

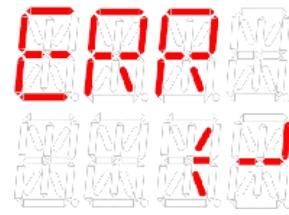
## 7.5.- Salida de error.

La salida de error está representada por el objeto `System.err`. Este objeto es también una instancia de la clase `PrintStream`, por lo que podemos utilizar los mismos métodos vistos anteriormente.

No parece muy útil utilizar `out` y `err` si su destino es la misma pantalla, o al menos en el caso de la consola del sistema donde las dos salidas son representadas con el mismo color y no notamos diferencia alguna. En cambio en la consola de varios entornos integrados de desarrollo como NetBeans o Eclipse la salida de `err` se ve en un color diferente. Teniendo el siguiente código:

```
System.out.println("Salida est&ardar por pantalla");
System.err.println("Salida de error por pantalla");
```

Mª Flor Moncada Añón ([CC BY-NC](#))



Ameya arsekar ([CC BY-SA](#))

La salida de este ejemplo en Netbeans es:

```
run:
Salida est&ardar por pantalla
Salida de error por pantalla
BUILD SUCCESSFUL (total time: 1 second)
```

Sun Microsystems. Captura de pantalla de la aplicación Netbeans. ([CC BY-NC](#))

Como vemos en un entorno como Netbeans, utilizar las dos salidas nos puede ayudar a una mejor depuración del código.

### Autoevaluaci&on

Relaciona cada clase con su funci&nacute;n, escribiendo el n&umero asociado a la funci&nacute;n en el hueco correspondiente.

#### Ejercicio de relacionar

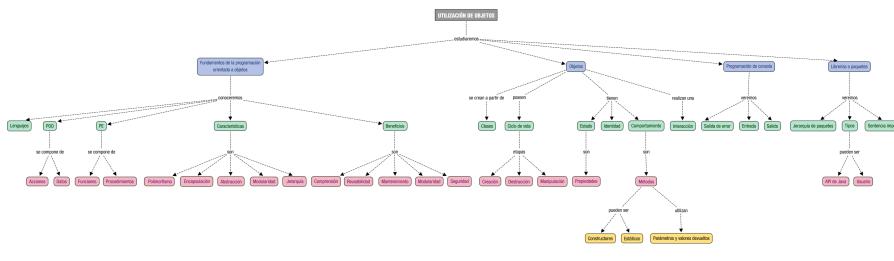
Clase.	Relaci&nacute;n.	Funci&nacute;n.
Scanner	<input type="checkbox"/>	1. Convierte los bytes leidos en caracteres.
PrintStream	<input type="checkbox"/>	2. Lee hasta un fin de línea.
InputStreamReader	<input type="checkbox"/>	3. Lee diferentes tipos de datos desde la consola de la aplicaci&nacute;n.
BufferedReader	<input type="checkbox"/>	4. Contiene varios métodos para mostrar datos por pantalla.

Enviar

PrintStream se utiliza para la salida de datos por pantalla y las otras clases, para la entrada de datos por teclado.

## 8.- Conclusiones.

A lo largo de esta tercera unidad, hemos introducido los fundamentos básicos sobre la programación orientada a objetos, el paradigma de programación utilizado por Java. Hemos aprendido el concepto de clase, de objeto, cómo se crean, manipulan y destruyen, cómo declaran y utilizan métodos y qué son paquetes o librerías de objetos. Por otro lado, hemos conocido las funcionalidades de Java para leer datos de teclado y mostrarlos por pantalla. En el mapa conceptual se pueden observar los conceptos que se deben tener claros antes de avanzar a la siguiente unidad.



Ministerio de Educación y FP ([CC BY-NC](#))

Llegados a este punto, estamos en disposición de conocer las estructuras de control del flujo de ejecución que proporciona el lenguaje Java. A partir de ese momento podremos construir aplicaciones Java mucho más funcionales. Será en la Unidad 4.



# Uso de estructuras de control.

## Caso práctico

Los miembros de BK Programación están inmersos en el mundo de Java, ya conocen una buena parte del lenguaje y ahora van a empezar a controlar el comportamiento de sus programas.

**María** pregunta a **Juan**: -Dime Juan, ¿En Java también hay condicionales y bucles?

-Efectivamente **María**, como la gran mayoría de los lenguajes de programación, Java incorpora estructuras que nos permiten tomar decisiones, repetir código, etc. Cada estructura tiene sus ventajas e inconvenientes y hay que saber dónde utilizar cada una de ellas. - Aclara **Juan**.



Ministerio de Educación y FP. ([CC BY-NC](#))



Ministerio de Educación y FP. ([CC BY-NC](#))

Stockbyte. CD-DVD Num. V43 ([CC BY-NC](#))



Ministerio de Educación y Formación Profesional. (Dominio público)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**  
[Aviso Legal](#)

# 1.- Introducción.

En unidades anteriores has podido aprender cuestiones básicas sobre el lenguaje Java: definición de variables, tipos de datos, asignación de valores, uso de literales, diferentes operadores que se pueden aplicar, conversiones de tipos, inserción de comentarios, etc. Posteriormente, nos sumergimos de lleno en el mundo de los objetos. Primero hemos conocido su filosofía, para más tarde ir recorriendo los conceptos y técnicas más importantes relacionadas con ellos: propiedades, métodos, clases, declaración y uso de objetos, librerías, etc.

Vale, parece ser que tenemos los elementos suficientes para comenzar a generar programas escritos en Java, ¿Seguro?

## Reflexiona

Piensa en la siguiente pregunta: ¿Cómo un programa puede determinar la aparición en pantalla de un mensaje de ÉXITO o ERROR, según los datos de entrada aportados por un usuario?

[Mostrar respuesta](#)

No son sólo los datos de entrada aportados por un usuario, existen más variables. Sigue con atención la unidad y lo comprenderás.

Como habrás deducido, con lo que sabemos hasta ahora no es suficiente. Existen múltiples situaciones que nuestros programas deben representar y que requieren tomar ciertas decisiones, ofrecer diferentes alternativas o llevar a cabo determinadas operaciones repetitivamente para conseguir sus objetivos.

Si has programado alguna vez o tienes ciertos conocimientos básicos sobre lenguajes de programación, sabes que la gran mayoría de lenguajes poseen estructuras que permiten a los programadores controlar el flujo de la información de sus programas. Esto realmente es una ventaja para la persona que está aprendiendo un nuevo lenguaje, o tienen previsto aprender más de uno, ya que estas estructuras suelen ser comunes a todos (con algunos cambios de sintaxis). Es decir, si conocías sentencias de control de flujo en otros lenguajes, lo que vamos a ver a lo largo de esta unidad te va a sonar bastante.



Stockbyte CD-DVD Num. V43  
(CC BY-NC)

Para alguien que no ha programado nunca, un ejemplo sencillo le va a permitir entender qué es eso de las sentencias de control de flujo. Piensa en un fontanero (programador), principalmente trabaja con agua (datos) y se encarga de hacer que ésta fluya por donde él quiere (programa) a través de un conjunto de tuberías, codos, latiguillos, llaves de paso, etc. (sentencias de control de flujo). Pues esas estructuras de control de flujo son las que estudiaremos, conoceremos su estructura, funcionamiento, cómo utilizarlas y dónde. A través de ellas, al construir nuestros programas podremos hacer que los datos (agua) fluyan por los caminos adecuados para representar la realidad del problema y obtener un resultado adecuado.

Los tipos de **estructuras** de programación que se emplean **para el control del flujo** de los datos son las siguientes:

- ✓ **Secuencia:** compuestas por 0, 1 o N sentencias que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.
- ✓ **Selección:** es un tipo de sentencia especial de decisión y de un conjunto de secuencias de instrucciones asociadas a ella. Según la evaluación de la sentencia de decisión se generará un resultado (que suele ser verdadero o falso) y en función de éste, se ejecutarán una secuencia de instrucciones u otra. Las estructuras de selección podrán ser simples, compuestas y múltiples.
- ✓ **Iteración:** es un tipo de sentencia especial de decisión y una secuencia de instrucciones que pueden ser repetidas según el resultado de la evaluación de la sentencia de decisión. Es decir, la secuencia de instrucciones se ejecutará repetidamente si la sentencia de decisión arroja un valor correcto, en otro caso la estructura de repetición se detendrá.

Además de las sentencias típicas de control de flujo, en esta unidad haremos una revisión de las **sentencias de salto**, que aunque no son demasiado recomendables, es necesario conocerlas. Como nuestros programas podrán generar errores y situaciones especiales, echaremos un vistazo al **manejo de excepciones** en Java. Posteriormente, analizaremos la mejor manera de llevar a cabo las **pruebas** de nuestros programas y la **depuración** de los mismos. Y finalmente, aprenderemos a valorar y utilizar las herramientas de **documentación de programas**.

Vamos entonces a ponernos el mono de trabajo y a coger nuestra caja de herramientas, ja ver si no nos mojamos mucho!

## 2.- Sentencias y bloques.

### Caso práctico

**Ada** valora muy positivamente en un programador el orden y la pulcritud. Organizar correctamente el código fuente es de vital importancia cuando se trabaja en entornos colaborativos, en los que son varios los desarrolladores los que forman los equipos de programación. Por ello, incide en la necesidad de recordar a **Juan y María** las nociones básicas a la hora de escribir programas.

Este epígrafe lo utilizaremos para reafirmar cuestiones que son obvias y que en el transcurso de anteriores unidades se han dado por sabidas. Aunque, a veces, es conveniente recordar. Lo haremos como un conjunto de FAQ:

- ✓ **¿Cómo se escribe un programa sencillo?** Si queremos que un programa sencillo realice instrucciones o sentencias para obtener un determinado resultado, es necesario colocar éstas una detrás de la otra, exactamente en el orden en que deben ejecutarse.
- ✓ **¿Podrían colocarse todas las sentencias una detrás de otra, separadas por puntos y comas en una misma línea?** Claro que sí, pero no es muy recomendable. Cada sentencia debe estar escrita en una línea, de esta manera tu código será mucho más legible y la localización de errores en tus programas será más sencilla y rápida. De hecho, cuando se utilizan herramientas de programación, los errores suelen asociarse a un número o números de línea.
- ✓ **¿Puede una misma sentencia ocupar varias líneas en el programa?** Sí. Existen sentencias que, por su tamaño, pueden generar varias líneas. Pero siempre finalizarán con un punto y coma.
- ✓ **¿En Java todas las sentencias se terminan con punto y coma?** Efectivamente. Si detrás de una sentencia ha de venir otra, pondremos un punto y coma. Escribiendo la siguiente sentencia en una nueva línea. Pero en algunas ocasiones, sobre todo cuando utilizamos estructuras de control de flujo, detrás de la cabecera de una estructura de este tipo no debe colocarse punto y coma. No te preocupes, lo entenderás cuando analicemos cada una de ellas.
- ✓ **¿Qué es la sentencia nula en Java?** La sentencia nula es una línea que no contiene ninguna instrucción y en la que sólo existe un punto y coma. Como su nombre indica, esta sentencia no hace nada.
- ✓ **¿Qué es un bloque de sentencias?** Es un conjunto de sentencias que se encierra entre llaves y que se ejecutaría como si fuera una única orden. Sirve para agrupar sentencias y para clarificar el código. Los bloques de sentencias son utilizados en Java en la práctica totalidad de estructuras de control de flujo, clases, métodos, etc. La siguiente tabla muestra dos formas de construir un bloque de sentencias.

#### Bloque de sentencias 1

```
{sentencia1; sentencia2;...; sentencia N;}
```

#### Bloque de sentencias 2

```
{
    sentencia1;
    sentencia2;
    ...;
    sentenciaN;
}
```

- ✓ **¿En un bloque de sentencias, éstas deben estar colocadas con un orden exacto?** En ciertos casos sí, aunque si al final de su ejecución se obtiene el mismo resultado, podrían ocupar diferentes posiciones en nuestro programa.

### Debes conocer

Accede a los tres archivos que te ofrecemos a continuación y compara su código fuente. Verás que los tres obtienen el mismo resultado, pero la organización de las sentencias que los componen es diferente entre ellos.

### Sentencias y bloques.

Sentencias, bloques y diferentes organizaciones		
<a href="#">Sentencias en orden secuencial.</a>	<a href="#">Sentencias y declaraciones de variables.</a>	<a href="#">Sentencias, declaraciones y organización del código.</a>
En este primer archivo, las sentencias están colocadas en orden secuencial.	En este segundo archivo, se declaran al principio las variables necesarias. En Java no es imprescindible hacerlo así, pero sí que antes de utilizar cualquier variable ésta debe estar previamente declarada. Aunque la declaración de dicha variable puede hacerse en cualquier lugar de nuestro programa.	En este tercer archivo, podrás apreciar que se ha organizado el código en las siguientes partes: declaración de variables, petición de datos de entrada, procesamiento de dichos datos y obtención de la salida. Este tipo de organización está más estandarizada y hace que nuestros programas ganen en legibilidad.

Construyas de una forma o de otra tus programas, debes tener en cuenta siempre en Java las siguientes premisas:

- ✓ Declara cada variable antes de utilizarla.
- ✓ Las variables, declaradas dentro de un método, no se inicializan con un valor, con lo cual, antes de trabajar con su valor, hay que darlas un valor inicial bien pidiendo ese dato por el teclado, dándole un valor concreto, un valor como resultado de una operación,... En el caso de las variables declaradas fuera de los métodos (datos de la clase) se les da un valor inicial, si el programador no se lo da:
  - ◆ A las variables numéricas le asigna un cero.
  - ◆ A los objetos se le asigna el valor `null`.
  - ◆ A las booleanas se le asigna el valor `false`.
  - ◆ A las variables de tipo `char` el valor ''.
- ✓ **No se deben usar variables no inicializadas en nuestros programas, pueden provocar errores o resultados imprevistos.**
- ✓ Utiliza sentencias que te permitan minimizar el código pero que sea fácilmente legible.

## Autoevaluación

Indica qué afirmación es correcta:

- Para crear un bloque de sentencias, es necesario delimitar éstas entre llaves. Este bloque funcionará como si hubiéramos colocado una única orden.
- La sentencia nula en Java, se puede representar con un punto y coma sólo en una única línea.
- Para finalizar en Java cualquier sentencia, es necesario hacerlo con un punto y coma.
- Todas las afirmaciones son correctas.

Incorrecto. Aunque la afirmación es correcta, deberías revisar el resto de respuestas.

Incorrecto. Aunque la afirmación es correcta, deberías revisar el resto de respuestas.

Incorrecto. Aunque la afirmación es correcta, deberías revisar el resto de respuestas.

Correcto. Todas y cada una de las afirmaciones de esta pregunta son correctas. Las sentencias es imprescindible que acaben en punto y coma, para indicar una sentencia nula basta con incluir una línea

con un punto y coma, y los bloques de sentencias se flanquean con llaves.

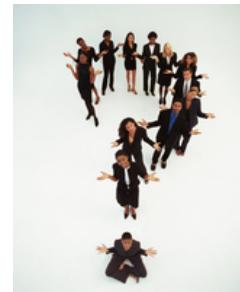
## Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

## 3.- Estructuras de selección.

### Caso práctico

Juan está desarrollando un método en el que ha de comparar los valores de las entradas de un usuario y una contraseña introducidas desde el teclado, con los valores almacenados en una base de datos. Para poder hacer dicha comparación necesitará utilizar una estructura condicional que le permita llevar a cabo esta operación, incluso necesitará que dicha estructura condicional sea capaz de decidir qué hacer en función de si ambos valores son correctos o no.



Stockbyte CD-DVD Num. CD109 [CC BY-NC](#)

Al principio de la unidad nos hacíamos esta pregunta: ¿Cómo un programa puede determinar la aparición en pantalla de un mensaje de ÉXITO o ERROR, según los datos de entrada aportados por un usuario? Esta y otras preguntas se nos plantean en múltiples ocasiones cuando desarrollamos programas.

¿Cómo conseguimos que nuestros programas puedan tomar decisiones? Para comenzar, lo haremos a través de las estructuras de selección. Estas estructuras constan de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones.

El funcionamiento es sencillo, la sentencia de decisión será evaluada y ésta devolverá un valor (verdadero o falso), en función del valor devuelto se ejecutará una secuencia de instrucciones u otra. Por ejemplo, si el valor de una variable es mayor o igual que 5 se imprime por pantalla la palabra APROBADO y, si es menor, se imprime SUSPENSO. Para este ejemplo, la comprobación del valor de la variable será la sentencia especial de decisión. La impresión de la palabra APROBADO será una secuencia de instrucciones y la impresión de la palabra SUSPENSO será otra. Cada secuencia estará asociada a cada uno de los resultados que puede arrojar la evaluación de la sentencia especial de decisión.

### Recomendación

En el lenguaje de programación C, verdadero o falso se representan mediante un literal entero. 0 representará Falso y 1 o cualquier otro valor, representará Verdadero. **Como sabes, en Java las variables de tipo booleano sólo podrán tomar los valores true (verdadero) o false (falso).** También existe la clase Boolean, no debemos confundirla con el tipo primitivo boolean.

La evaluación de las sentencias de decisión o expresiones que controlan las estructuras de selección, devolverán siempre un valor verdadero o falso.

Las estructuras de selección se dividen en:

1. Estructuras de selección simples o estructura if.
2. Estructuras de selección compuestas o estructura if-else.
3. Estructuras de selección basadas en el operador condicional.
4. Estructuras de selección múltiples o estructura switch.

A continuación, detallaremos las características y funcionamiento de cada una de ellas. Es importante que a través de los ejemplos que vamos a ver, puedas determinar en qué circunstancias utilizar cada una de estas estructuras. Aunque un mismo problema puede ser resuelto con diferentes estructuras e incluso, con diferentes combinaciones de éstas.

## 3.1.- Estructura if / if-else.

---

La estructura `if` es una estructura de selección o estructura condicional, en la que se evalúa una expresión lógica o sentencia de decisión y en función del resultado, se ejecuta una sentencia o un bloque de éstas.

La estructura `if` puede presentarse de las siguientes formas:

### Estructura if simple

```
if (expresión-lógica) //al ser una sola instrucción, no son necesarias las llaves  
sentencia1;
```

```
if (expresión-lógica)  
{  
    sentencia1;  
    sentencia2;  
    ...;  
    sentenciaN;  
}
```

Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecuta la `sentencia1` o bien el bloque de sentencias asociado. Si el resultado de dicha evaluación es falso, no se ejecutará ninguna instrucción asociada a la estructura condicional.

### Estructura if de doble alternativa

```
if (expresión-lógica) //al ser una sola instrucción, no son necesarias las llaves  
sentencia1;  
else  
    sentencia2;  
  
if (expresión-lógica)  
{  
    sentencia1;  
    ...;  
    sentenciaN;  
}  
else  
{  
    sentencia1;  
    ...;  
    sentenciaN;  
}
```

Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecutará la primera sentencia o el primer bloque de sentencias. Si, por el contrario, la evaluación de la expresión-lógica ofrece un resultado falso, no se ejecutará la primera sentencia o el primer bloque y sí se ejecutará la segunda sentencia o el segundo bloque.

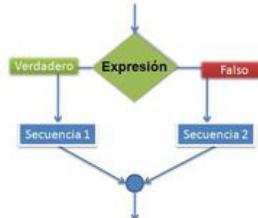
Haciendo una interpretación cercana al pseudocódigo tendríamos que si se cumple la condición (expresión lógica), se ejecutará un conjunto de instrucciones y si no se cumple, se ejecutará otro conjunto de instrucciones.

Hay que tener en cuenta que la cláusula `else` de la sentencia `if` no es obligatoria. En algunos casos no necesitaremos utilizarla, pero sí se recomienda cuando es necesario llevar a cabo alguna acción en el caso de que la expresión lógica no se cumpla.

En aquellos casos en los que no existe cláusula `else`, si la expresión lógica es falsa, simplemente se continuarán ejecutando las siguientes sentencias que aparezcan bajo la estructura condicional `if`.

Los condicionales `if` e `if-else` pueden anidarse, de tal forma que dentro de un bloque de sentencias puede incluirse otro `if` o `if-else`. El nivel de anidamiento queda a criterio del programador, pero si éste es demasiado profundo podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede denotar la necesidad de utilización de otras estructuras de selección más adecuadas.

Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué `if` está asociada una cláusula `else`. Normalmente, un `else` estará asociado con el `if` inmediatamente superior o más cercano que exista dentro del mismo bloque y que no se encuentre ya asociado a otro `else`.



José Luis García Martínez. [\(CC BY-NC\)](#)

## Debes conocer

Para completar la información que debes saber sobre las estructuras `if` e `if-else`, accede al siguiente enlace. En él podrás analizar el código de un programa que realiza el cálculo de la nota de un examen de tipo test. Además de calcular el valor de la nota, se ofrece como salida la calificación no numérica de dicho examen. Para obtenerla, se combinarán las diferentes estructuras condicionales aprendidas hasta ahora.

Presta especial atención a los comentarios incorporados en el código fuente, así como a la forma de combinar las estructuras condicionales y a las expresiones lógicas utilizadas en ellas.

[Uso la estructura condicional if e if-else. \(0.01 MB\)](#)

## Autoevaluación

¿Cuándo se mostrará por pantalla el mensaje incluido en el siguiente fragmento de código?

```
If (numero % 2 == 0);  
System.out.print("El número es par /n");
```

- Nunca.
- Siempre.
- Cuando el resto de la división entre 2 del contenido de la variable `numero`, sea cero.

Incorrecto. Aunque detrás de la expresión lógica de la estructura `if` existe un punto y coma, la sentencia que imprime el mensaje en pantalla siempre se ejecutará.

Efectivamente. Aunque detrás de la expresión lógica de la estructura if existe un punto y coma, la sentencia que imprime el mensaje en pantalla siempre se ejecutará.

Incorrecto. El punto y coma colocado justo detrás de la expresión lógica de la estructura if representaría la sentencia vacía, y por tanto, la impresión del mensaje no quedaría asociada al resultado de la evaluación de dicha expresión lógica.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

## 3.2.- Estructura switch.

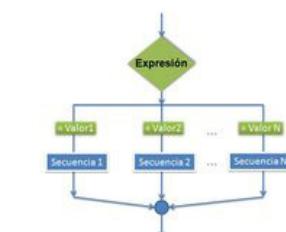
¿Qué podemos hacer cuando nuestro programa debe elegir entre más de dos alternativas?. Una posible solución podría ser emplear estructuras `if` anidadas, aunque no siempre esta solución es la más eficiente. Cuando estamos ante estas situaciones podemos utilizar la estructura de selección múltiple `switch`. En la siguiente tabla se muestra tanto la sintaxis, como el funcionamiento de esta estructura.

```
switch (expresión) {  
    case valor1:  
        sentencia1_1;  
        sentencia1_2;  
        ...  
        break;  
        ...  
        ...  
    case valorN:  
        sentenciaN_1;  
        sentenciaN_2;  
        ...  
        break;  
    default:  
        sentencias-default;  
}
```

### Condiciones:

- ✓ Donde expresión debe ser del tipo `char`, `byte`, `short` o `int`, y las constantes de cada case deben ser de este tipo o de un tipo compatible.
- ✓ La expresión debe ir entre paréntesis.
- ✓ Cada `case` llevará asociado un valor y se finalizará con dos puntos.
- ✓ El bloque de sentencias asociado a la cláusula `default` puede finalizar con una sentencia de ruptura `break` o no.

En resumen, se ha de comparar el valor de una expresión con un conjunto de constantes, si el valor de la expresión coincide con algún valor de dichas constantes, se ejecutarán los bloques de instrucciones asociados a cada una de ellas. Si no existiese coincidencia, se ejecutarían una serie de instrucciones por defecto.



José Luis García Martínez. ([CC BY-NC](#))

**Debes conocer**

Accede al siguiente fragmento de código en el que se resuelve el cálculo de la nota de un examen de tipo test, utilizando la estructura `switch`.

[Uso la estructura condicional múltiple `switch`.](#) (0.01 MB)

## 4.- Estructuras de repetición.

### Caso práctico

Juan ya tiene claro cómo realizar la comprobación de los valores de usuario y contraseña introducidos por teclado, pero le surge una duda: ¿Cómo podría controlar el número de veces que el usuario ha introducido mal la contraseña?

Ada le indica que podría utilizar una estructura de repetición que solicitase al usuario la introducción de la contraseña hasta un máximo de tres veces. Aunque comenta que puede haber múltiples soluciones y todas válidas, lo importante es conocer las herramientas que podemos emplear y saber cuándo aplicarlas.

Nuestros programas ya son capaces de controlar su ejecución teniendo en cuenta determinadas condiciones, pero aún hemos de aprender un conjunto de estructuras que nos permita repetir una secuencia de instrucciones determinada. La función de estas estructuras es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

A este tipo de estructuras se las denomina **estructuras de repetición**, estructuras repetitivas, **bucles** o estructuras **iterativas**. En Java existen cuatro clases de bucles:



HuBoro (Dominio público)

- ✓ Bucle for (repite para)
- ✓ Bucle for/in (repite para cada)
- ✓ Bucle While (repite mientras)
- ✓ Bucle Do While (repite hasta)

Los bucles **for** y **for/in** se consideran bucles **controlados por contador**. Por el contrario, los bucles **while** y **do...while** se consideran bucles **controlados por sucesos**.

La utilización de unos bucles u otros para solucionar un problema dependerá en gran medida de las siguientes preguntas:

- ✓ ¿Sabemos a priori cuántas veces necesitamos repetir un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de repetir un conjunto de instrucciones si una condición satisface un conjunto de valores?
- ✓ ¿Sabemos hasta cuándo debemos estar repitiendo un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de estar repitiendo un conjunto de instrucciones mientras se cumpla una condición?

### Recomendación

Estudia cada tipo de estructura repetitiva, conoce su funcionamiento y podrás llegar a la conclusión de que algunos de estos bucles son equivalentes entre sí. Un mismo problema, podrá ser resuelto empleando diferentes tipos de bucles y obtener los mismos resultados.

Como ya se ha recomendado y como habrás podido comprobar, la mejor de aprender es programar los ejemplos en Netbeans, corregir fallos y lanzarlos a ejecución. ¡Mucho ánimo!

Estas y otras preguntas tendrán su respuesta en cuanto analicemos cada una de estructuras repetitivas en detalle.

## 4.1.- Estructura for.

Hemos indicado anteriormente que el bucle `for` es un bucle controlado por contador. Este tipo de bucle tiene las siguientes características:

- ✓ Se ejecuta un número determinado de veces.
- ✓ Utiliza una variable contadora que controla las iteraciones del bucle.

En general, existen tres operaciones que se llevan a cabo en este tipo de bucles:

- ✓ Se inicializa la variable contadora.
- ✓ Se evalúa el valor de la variable contadora, por medio de una comparación de su valor con el número de iteraciones especificado.
- ✓ Se modifica o actualiza el valor del contador a través de incrementos o decrementos de éste, en cada una de las iteraciones.



José Luis García Martínez (CC BY-NC)

### Recomendación

La inicialización de la variable contadora debe realizarse correctamente para garantizar que el bucle se lleve a cabo, al menos, la primera repetición de su código interno.

La condición de terminación del bucle debe variar en el interior del mismo, de no ser así, podemos caer en la creación de un bucle infinito. Cuestión que se debe evitar por todos los medios.

Es necesario estudiar el número de veces que se repite el bucle, pues debe ajustarse al número de veces estipulado.

En la siguiente tabla, podemos ver la especificación de la estructura `for`:

#### Bucle con una sola sentencia y con un bloque de sentencias

```
for (inicialización; condición; iteración)
    sentencia; //Con una sola instrucción no es necesario utilizar llaves

for (inicialización; condición; iteración)
{
    sentencia1;
    sentencia2;
    ...
    sentenciaN;
}
```

Donde:

- **inicialización** es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle.
- **condición** es una expresión que evaluará la variable de control. Mientras la condición sea falsa, el cuerpo del bucle estará repitiéndose. Cuando la condición se cumpla, terminará la ejecución del bucle.
- **iteración** indica la manera en la que la variable de control va cambiando en cada iteración del bucle. Podrá ser mediante incremento o decremento, y no solo de uno en uno.

## Debes conocer

Como venimos haciendo para el resto de estructuras, accede al siguiente archivo Java y podrás analizar un ejemplo de utilización del bucle for para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle.

[Uso la estructura repetitiva for.](#)

## Autoevaluación

Cuando construimos la cabecera de un bucle for, podemos prescindir de alguno de los tres elementos que la forman e incluso, podemos utilizar más de una variable contadora separando éstas por comas. Pero, ¿Qué conseguiremos si construimos un bucle de la siguiente forma?

```
for (;;) { //instrucciones }
```

- Un bucle infinito.
- Nada, dará un error.
- Un bucle que se ejecutaría una única vez.

Efectivamente. La construcción de la cabecera del bucle ha prescindido de cada una de las partes típicas (inicialización, condición e iteración), pero al mantener los puntos y comas en los lugares adecuados, el conjunto de instrucciones que pudieran incluirse en el cuerpo del bucle estarán ejecutándose eternamente.

Incorrecto. La sintaxis es válida y lo que estaríamos creando es un bucle infinito.

Incorrecto. Al no existir inicialización, condición ni iteración estaríamos construyendo un bucle infinito.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

## 4.2.- Estructura for/in.

Junto a la estructura `for`, `for/in` también se considera un bucle controlado por contador. Este bucle es una mejora incorporada en la versión 5.0. de Java.

Este tipo de bucles permite realizar recorridos sobre `arrays` y colecciones de objetos. Los `arrays` son colecciones de variables que tienen el mismo tipo y se referencian por un nombre común. Así mismo, las colecciones de objetos son objetos que se dice son iterables, o que se puede iterar sobre ellos.

Este bucle es nombrado también como bucle `for` mejorado, o bucle `foreach`. En otros lenguajes de programación existen bucles muy parecidos a este.

**La sintaxis es la siguiente:**

```
for (declaración: expresión) {  
    sentencia1;  
    ...  
    sentenciaN;  
}
```

- ✓ Donde `expresión` es un array o una colección de objetos.
- ✓ Donde `declaración` es la declaración de una variable cuyo tipo sea compatible con `expresión`. Normalmente, será el tipo y el nombre de la variable a declarar.

El funcionamiento consiste en que para cada elemento de la expresión, guarda el elemento en la variable declarada y realiza las instrucciones contenidas en el bucle. Después, en cada una de las iteraciones del bucle tendremos en la variable declarada el elemento actual de la expresión. Por tanto, para el caso de los `arrays` y de las colecciones de objetos, se recorrerá desde el primer elemento que los forma hasta el último.

Observa el contenido del código representado en la siguiente imagen, puedes apreciar cómo se construye un bucle de este tipo y su utilización sobre un `array`.

Los bucles `for/in` permitirán al programador despreocuparse del número de veces que se ha de iterar, pero no sabremos en qué iteración nos encontramos salvo que se añada artificialmente alguna variable contadora que nos pueda ofrecer esta información.



José Luis García Martínez (CC BY-NC)

## 4.3.- Estructura while.

El bucle `while` es la primera de las estructuras de repetición controladas por sucesos que vamos a estudiar. La utilización de este bucle responde al planteamiento de la siguiente pregunta: ¿Qué podemos hacer si lo único que sabemos es que se han de repetir un conjunto de instrucciones mientras se cumpla una determinada condición?

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle podría ser necesario ejecutarlas o no. Es decir, en el bucle `while` siempre se evaluará la condición que lo controla, y si dicha condición es cierta, el cuerpo del bucle se ejecutará una vez, y se seguirá ejecutando mientras la condición sea cierta. Pero si en la evaluación inicial de la condición ésta no es verdadera, el cuerpo del bucle no se ejecutará.

Es imprescindible que en el interior del bucle `while` se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

### Bucle while con una sentencia y un bloque de sentencias

```
while (condición)
```

```
    sentencia;
```

```
while (condición) {
```

```
    sentencia1;
```

```
    ...
```

```
    sentenciaN;
```

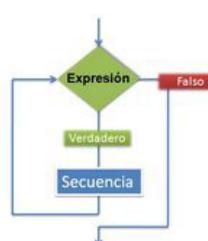
```
}
```

### Funcionamiento:

Mientras la condición sea cierta, el bucle se repetirá, ejecutando la/s instrucción/es de su interior. En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle `while`.

La condición se evaluará siempre al principio, y podrá darse el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca si no se satisface la condición de partida.

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.



José Luis García Martínez. ([CC BY-NC](#))

### Debes conocer

Accede al siguiente archivo java y podrás analizar un ejemplo de utilización del bucle `while` para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle `for`.

## Autoevaluación

Utilizando el siguiente fragmento de código estamos construyendo un bucle infinito. ¿Verdadero o Falso?

```
while (true) System.out.println("Imprimiendo desde dentro del bucle \n");
```

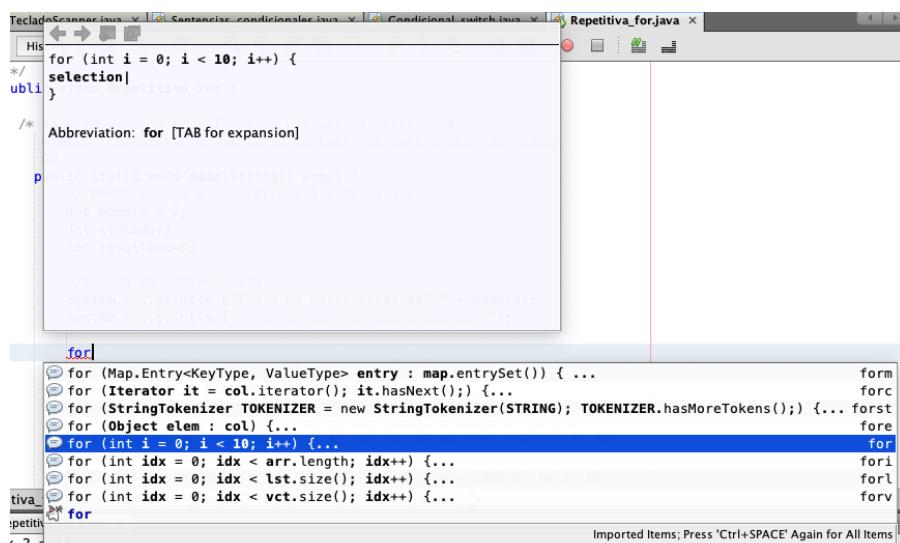
- Verdadero  Falso

**Verdadero**

La condición del bucle siempre será cierta y estará continuamente ejecutándose.

## Debes conocer

Siguiendo con las ayudas de Netbeans para la inserción de código, para incluir un bucle también podemos recurrir a la genialidad del entorno para insertar código automáticamente. Escribe **for** y pulsa **Ctrl + Espacio** en el editor. Verás algo parecido a lo que se muestra en la imagen: diferentes opciones de construcción del bucle **for**. Si seleccionas una, Netbeans automáticamente incluirá el código.



## 4.4.- Estructura do-while.

La segunda de las estructuras repetitivas controladas por sucesos es **do-while**. En este caso, la pregunta que nos planteamos es la siguiente: ¿Qué podemos hacer si lo único que sabemos es que se han de ejecutar, al menos una vez, un conjunto de instrucciones y seguir repitiéndose hasta que se cumpla una determinada condición?

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle necesitan ser ejecutadas, al menos, una vez y repetir su ejecución hasta que la condición sea verdadera. Por tanto, en esta estructura repetitiva siempre se ejecuta el cuerpo del bucle una primera vez.

Es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

### Estructura do-while con una sentencia

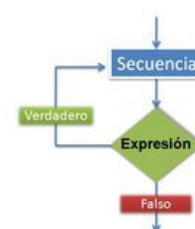
```
do
    sentencia; //Con una sola sentencia no son necesarias las llaves
    while (condición);
```

### Estructura do-while con un bloque de sentencias

```
do
{
    sentencia1;
    ...
    sentenciaN;
}
while (condición);
```

El cuerpo del bucle se ejecuta la primera vez, a continuación se evaluará la condición y, si ésta es falsa, el cuerpo del bucle volverá a repetirse. El bucle finalizará cuando la evaluación de la condición sea verdadera. En ese momento el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle do-while. La condición se evaluará siempre después de una primera ejecución del cuerpo del bucle, por lo que no se dará el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca.

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.



José Luis García Martínez ([CC BY-NC](#))

### Debes conocer

Accede al siguiente archivo java y podrás analizar un ejemplo de utilización del bucle **do-while** para la impresión por pantalla de la tabla de multiplicar del siete. Lee atentamente los comentarios incluidos en el código, pues aclaran algunas cuestiones interesantes sobre este bucle. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle **for** y el bucle **while**.

[Uso la estructura repetitiva do-while.](#) (0.01 MB)

## 5.- Estructuras de salto.

### Caso práctico

**Juan** recuerda que algunos lenguajes de programación permitían realizar saltos a lo largo de la ejecución de los programas y conoce dos sentencias que aún se siguen utilizando para ello.

**Ada**, mientras toma un libro sobre Java de la estantería del despacho, le aconseja: -las instrucciones de salto a veces han sido mal valoradas por la comunidad de programadores, pero en Java algunas de ellas son totalmente necesarias. Mira, en este libro se habla del uso de las sentencias `break`, `continue` y `return`.

¿Saltar o no saltar? he ahí la cuestión. En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer y que pueden sernos útiles en algunas partes de nuestros programas.

Estas estructuras de salto corresponden a las sentencias `break`, `continue`, las **etiquetas de salto** y la sentencia `return`. Pasamos ahora a analizar su sintaxis y funcionamiento.

## 5.1.- Sentencias break y continue.

Se trata de dos instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.

La **sentencia** `break` incidirá sobre las estructuras de control `switch`, `while`, `for` y `do-while` del siguiente modo:

- ✓ Si aparece una sentencia `break` dentro de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha estructura terminará inmediatamente.
  - ✓ Si aparece una sentencia `break` dentro de un bucle anidado sólo finalizará la sentencia de iteración más interna, el resto se ejecuta de forma normal.

Es decir, que `break` sirve para romper el flujo de control de un bucle, aunque no se haya cumplido la condición del bucle. Si colocamos un `break` dentro del código de un bucle, cuando se alcance el `break`, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.

En la siguiente imagen, puedes apreciar cómo se utilizaría la sentencia `break` dentro de un bucle `for`.

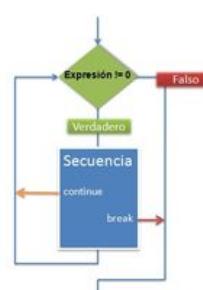
La **sentencia continue** incidirá sobre las sentencias o estructuras de control `while`, `for` y `do-while` del siguiente modo:

- ✓ Si aparece una sentencia continua dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.
  - ✓ Si aparece en el interior de un bucle anidado solo afectará a la sentencia de iteración más interna, el resto se ejecutaría de forma normal.

Es decir, la sentencia `continue` forzará a que se ejecute la siguiente iteración del bucle, sin tener en cuenta las instrucciones que pudiera haber después del `continue`, y hasta el final del código del bucle.

En la siguiente imagen, puedes apreciar cómo se utiliza la sentencia continue en un bucle `for` para imprimir por pantalla sólo los números pares.

Para clarificar algo más el funcionamiento de ambas sentencias de salto, te ofrecemos a continuación un diagrama representativo.



Pedro Fernández Arias (CC BY-NC)

## Autoevaluación

La instrucción `break` puede utilizarse en las estructuras de control `switch`, `while`, `for` y `do-while`, no siendo imprescindible utilizarla en la cláusula `default` de la estructura `switch`. ¿Verdadero o Falso?

Verdadero  Falso

**Verdadero**

Correcto, si en una cláusula `default` no existe una sentencia `break`, directamente se pasaría a la finalización

de la estructura condicional múltiple.

## 5.2.- Sentencia Return.

Ya sabemos como modificar la ejecución de bucles y estructuras condicionales múltiples, pero ¿Podríamos modificar la ejecución de un método? ¿Es posible hacer que éstos detengan su ejecución antes de que finalice el código asociado a ellos? Sí es posible, a través de la sentencia `return` podremos conseguirlo.



La sentencia `return` puede utilizarse de dos formas:

- ✓ Para terminar la ejecución del método donde esté escrita, con lo que transferirá el control al punto desde el que se hizo la llamada al método, continuando el programa por la sentencia inmediatamente posterior.
- ✓ Para devolver o retornar un valor, siempre que junto a `return` se incluya una expresión de un tipo determinado. Por tanto, en el lugar donde se invocó al método se obtendrá el valor resultante de la evaluación de la expresión que acompañaba al método.

En general, una sentencia `return` suele aparecer al final de un método, de este modo el método tendrá una entrada y una salida. **También es posible utilizar una sentencia `return` en cualquier punto de un método, con lo que éste finalizará en el lugar donde se encuentre dicho `return`.** No será recomendable incluir más de un `return` en un método y por regla general, deberá ir al final del método, como hemos comentado.

**El valor de retorno es opcional, si lo hubiera debería de ser del mismo tipo o de un tipo compatible al tipo del valor de retorno definido en la cabecera del método,** pudiendo ser desde un entero a un objeto creado por nosotros. Si no lo tuviera, el tipo de retorno sería `void`, y `return` serviría para salir del método sin necesidad de llegar a ejecutar todas las instrucciones que se encuentran después del `return`.

### Para saber más

En el siguiente archivo java encontrarás el código de un programa que obtiene la suma de dos números, empleando para ello un método sencillo que retorna el valor de la suma de los números que se le han pasado como parámetros. Presta atención a los comentarios y fíjate en las conversiones a entero de la entrada de los operandos por consola.

[Uso de `return` en métodos.](#) (0.01 MB)

### Autoevaluación

#### ¿Qué afirmación es correcta?

- Con `return`, se puede finalizar la ejecución del método en el que se encuentre.
- Con `return`, siempre se retornará un valor del mismo tipo o de un tipo compatible al definido en la cabecera del método.
- Con `return`, puede retornarse un valor de un determinado tipo y suele hacerse al final del método. Además, el resto de respuestas también son correctas.

Efectivamente, si encontramos un `return` antes de la finalización del código asociado a un método, éste devolverá el control del flujo del programa a la instrucción inmediatamente posterior que existiese tras la invocación al método.

Incorrecto, no siempre se retorna un valor. `Return` puede ir o no acompañado por una expresión.

Incorrecto, `return` puede retornar un valor de un tipo y esto suele hacerse al final del método, pero no todas las respuestas son correctas.

## **Solución**

1. Opción correcta
2. Incorrecto
3. Incorrecto

## 6.- Excepciones.

### Caso práctico

Para que las aplicaciones desarrolladas por BK Programación mantengan una valoración positiva a lo largo del tiempo por parte de sus clientes, es necesario que éstas no se vean comprometidas durante su ejecución. Hay innumerables ocasiones en las que programas que aparentemente son formidables (por su interfaz, facilidad de uso, etc.) comienzan a generar errores en tiempo de ejecución que hacen que el usuario desconfíe de ellos día a día.

Para evitar estas situaciones, Ada va a fomentar en María y Juan la cultura de la detección, control y solución de errores a través de las poderosas herramientas que Java les ofrece.



Stockbyte CD-DVD Num. V43 ([CC BY-NC](#))

A lo largo de nuestro aprendizaje de Java nos hemos topado en alguna ocasión con Errores, pero éstos suelen ser los que nos ha indicado el compilador. Un punto y coma por aquí, un nombre de variable incorrecto por allá, pueden hacer que nuestro compilador nos avise de estos descuidos. Cuando los vemos, se corrigen y obtenemos nuestra clase compilada correctamente.

Pero, ¿Sólo existen este tipo de Errores? ¿Podrían existir Errores no sintácticos en nuestros programas? Está claro que sí, un programa perfectamente compilado en el que no existen Errores de sintaxis, puede generar otros tipos de Errores que quizás aparezcan en tiempo de ejecución. A estos Errores se les conoce como **excepciones**.

Aprenderemos a gestionar de manera adecuada estas excepciones y tendremos la oportunidad de utilizar el potente sistema de manejo de Errores que Java incorpora. La potencia de este sistema de manejo de Errores radica en:

1. Que el código que se encarga de manejar los Errores, es perfectamente identificable en los programas. Este código puede estar separado del código que maneja la aplicación.
2. Que Java tiene una gran cantidad de Errores estándar asociados a multitud de fallos comunes, como por ejemplo divisiones por cero, fallos de entrada de datos, etc. Al tener tantas excepciones localizadas, podemos gestionar de manera específica cada uno de los Errores que se produzcan.

En Java se pueden preparar los fragmentos de código que pueden provocar Errores de ejecución para que si se produce una excepción, el flujo del programa es lanzado (*throw*) hacia ciertas zonas o rutinas que han sido creadas previamente por el programador y cuya finalidad será el tratamiento efectivo de dichas excepciones. Si no se captura la excepción, el programa se detendrá con toda probabilidad.

En Java, las excepciones están representadas por clases. El paquete `java.lang.Exception` y sus subpaquetes contienen todos los tipos de excepciones. Todas las excepciones derivarán de la clase `Throwable`, existiendo clases más específicas. Por debajo de la clase `Throwable` existen las clases `Error` y `Exception`. `Errors` una clase que se encargará de los Errores que se produzcan en la máquina virtual, no en nuestros programas. Y la clase `Exception` será la que a nosotros nos interese conocer, pues gestiona los Errores provocados en los programas.

Java lanzará una excepción en respuesta a una situación poco usual. Cuando se produce un `Error` se genera un objeto asociado a esa excepción. Este objeto es de la clase `Exception` o de alguna de sus herederas. Este objeto se pasa al código que se ha definido para manejar la excepción. Dicho código puede manipular las propiedades del objeto `Exception`.

El programador también puede lanzar sus propias excepciones. Las excepciones en Java serán objetos de clases derivadas de la clase base `Exception`. Existe toda una jerarquía de clases derivada de la clase base `Exception`. Estas clases derivadas se ubican en dos grupos principales:

- ✓ Las excepciones en tiempo de ejecución, que ocurren cuando el programador no ha tenido cuidado al escribir su código.
- ✓ Las excepciones que indican que ha sucedido algo inesperado o fuera de control.

En la siguiente imagen te ofrecemos una aproximación a la jerarquía de las excepciones en Java.



Erik van Leeuwen ([GNU/GPL](#))



José Luis García Martínez ([CC BY-NC](#))

## 6.1.- Capturar una excepción.

Para poder capturar excepciones, emplearemos la estructura de captura de excepciones `try-catch-finally`.

Básicamente, para capturar una excepción lo que haremos será declarar bloques de código donde es posible que ocurra una excepción. Esto lo haremos mediante un bloque `try` (intentar). Si ocurre una excepción dentro de estos bloques, se lanza una excepción. Estas excepciones lanzadas se pueden capturar por medio de bloques `catch`. Será dentro de este tipo de bloques donde se hará el manejo de las excepciones.

Su sintaxis es:

```
try {  
    código que puede generar excepciones;  
}  
} catch (Tipo_excepcion_1 objeto_excepcion) {  
    Manejo de excepción de Tipo_excepcion_1;  
}  
} catch (Tipo_excepcion_2 objeto_excepcion) {  
    Manejo de excepción de Tipo_excepcion_2;  
}  
}  
...  
finally {  
    instrucciones que se ejecutan siempre  
}
```

En esta estructura, la parte `catch` puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte `finally` es opcional y, si aparece, solo podrá hacerlo una sola vez.

Cada `catch` maneja un tipo de excepción. Cuando se produce una excepción, se busca el `catch` que posea el manejador de excepción adecuado, será el que utilice el mismo tipo de excepción que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la clase `Exception` es la superclase de todas las demás. Por lo que si se produjo, por ejemplo, una excepción de tipo `AritmeticException` y el primer `catch` captura el tipo genérico `Exception`, será ese `catch` el que se ejecute y no los demás.

Por eso el último `catch` debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo `catch` que capture objetos `Exception`.

### Ejercicio resuelto

Realiza un programa en Java en el que se solicite al usuario la introducción de un número por teclado comprendido entre el 0 y el 100. Utilizando manejo de excepciones, debes controlar la entrada de dicho número y volver a solicitarlo en caso de que ésta sea incorrecta.

[Mostrar retroalimentación](#)

Accede al siguiente enlace, en el que podrás visualizar cómo podríamos obtener los resultados solicitados en el enunciado del ejercicio, utilizando estructuras `try-catch-finally`.

[Filtrado de entrada desde teclado a través de excepciones. \(0.01\)](#)

En este programa se solicita repetidamente un número utilizando una estructura `do-while`, mientras el número introducido sea menor que 0 y mayor que 100. Como al solicitar el número pueden producirse los errores siguientes:

- ✓ De entrada de información a través de la excepción `IOException` generada por el método `readLine()` de la clase `BufferedReader`.
- ✓ De conversión de tipos a través de la excepción `NumberFormatException` generada por el método `parseInt()`.

Entonces se hace necesaria la utilización de bloques `catch` que gestionen cada una de las excepciones que puedan producirse. Cuando se produce una excepción, se compara si coincide con la excepción del primer `catch`. Si no coincide, se compara con la del segundo `catch` y así sucesivamente. Si se encuentra un `catch` que coincide con la excepción a gestionar, se ejecutará el bloque de sentencias asociado a éste.

Si ningún bloque `catch` coincide con la excepción lanzada, dicha excepción se lanzará fuera de la estructura `try-catch-finally`.

El bloque `finally`, se ejecutará tanto si `try` terminó correctamente, como si se capturó una excepción en algún bloque `catch`. Por tanto, si existe bloque `finally` éste se ejecutará siempre.

## Debes conocer

Si deseas conocer en mayor detalle el manejo de excepciones, te aconsejamos el siguiente enlace en el que podrás encontrar un vídeo ilustrativo sobre su creación y manejo.

**Excepciones en Java.**

<https://www.youtube.com/embed/2gWTVxe31g8>

*Enlace a video de youtube donde se ejemplifica el uso de excepciones.*

[Resumen textual alternativo](#)

## Autoevaluación

**Si en un programa no capturamos una excepción, será la máquina virtual de Java la que lo hará por nosotros, pero inmediatamente detendrá la ejecución del programa y mostrará una traza y un mensaje de error. Siendo una traza, la forma de localizar dónde se han producido errores. ¿Verdadero o Falso?**

Verdadero  Falso

### Verdadero

Así es, si un método no gestiona las excepciones que se producen en él, la excepción será pasada al método que llamó al método. Y así ocurrirá sucesivamente si no existe gestión de las excepciones en niveles superiores, hasta llegar a la máquina virtual de Java que detendrá la ejecución del programa.

## 6.2.- El manejo de excepciones.

Como hemos comentado, siempre debemos controlar las excepciones que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Las excepciones pueden tratarse de dos formas:

- ✓ **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y se entiende que no hay manera de regresar al código que provocó la excepción. Es decir, la operación que originó el error, se anula.
- ✓ **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

Java emplea la primera forma, pero puede simularse la segunda mediante la utilización de un bloque `try` en el interior de un `while`, que se repetirá hasta que el error deje de existir. En la siguiente imagen tienes un ejemplo de cómo llevar a cabo esta simulación.



Pedro Fernández Arias ([CC BY-NC](#))

En este ejemplo, a través de la función de generación de números aleatorios se obtiene el valor del índice `i`. Con dicho valor se accede a una posición del array que contiene cinco cadenas de caracteres. Este acceso, a veces puede generar un error del tipo `ArrayIndexOutOfBoundsException`, que debemos gestionar a través de un `catch`. Al estar el bloque `catch` dentro de un `while`, se seguirá intentando el acceso hasta que no haya error.

## 6.3.- Delegación de excepciones con throws.

¿Puede haber problemas con las excepciones al usar llamadas a métodos en nuestros programas? Efectivamente, si se produjese una excepción es necesario saber quién será el encargado de solucionarla. Puede ser que sea el propio método llamado o el código que hizo la llamada a dicho método.

Quizá pudiéramos pensar que debería ser el propio método el que se encargue de sus excepciones, aunque es posible hacer que la excepción sea resuelta por el código que hizo la llamada. Cuando un método utiliza una sentencia que puede generar una excepción, pero dicha excepción no es capturada y tratada por él, sino que se encarga su gestión a quién llamó al método, decimos que se ha producido **delegación de excepciones**.

Para establecer esta delegación, en la cabecera del método se declara el tipo de excepciones que puede generar y que deberán ser gestionadas por quien invoque a dicho método. Utilizaremos para ello la sentencia `throws` y tras esa palabra se indica qué excepciones puede provocar el código del método. Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el `catch` apropiado para esa excepción. Su sintaxis es la siguiente:

```
public class delegacion_excepciones {  
    ...  
    public int leeño(Reader lector) throws IOException, NumberFormatException{  
        String linea = teclado.readLine();  
        Return Integer.parseInt(linea);  
    }  
    ...  
}
```

Donde `IOException` y `NumberFormatException`, serían dos posibles excepciones que el método `leeño` podría generar, pero que no gestiona. Por tanto, un método puede incluir en su cabecera un listado de excepciones que puede lanzar, separadas por comas.

### Para saber más

Si deseas saber algo más sobre la delegación de excepciones, te proponemos el siguiente enlace:

[Excepciones y delegación de éstas.](#)

Además te volvemos a remitir al vídeo demostrativo sobre manejo de excepciones en Java que se incluyó en el epígrafe anterior, titulado "capturar una excepción".



Stockbyte CD-DVD Num. V43 (CC BY-NC)

## 7.- Depuración de programas.

### Caso práctico

**Ada y Juan** ya conocen las capacidades del depurador que incorpora el entorno NetBeans y van a enseñar a **María** las ventajas de utilizarlo.

-Puedes depurar tus programas haciendo dos cosas: creando Breakpoints o haciendo ejecuciones paso a paso -Le comenta **Juan**.

**María**, que estaba codificando un nuevo método, se detiene un momento y pregunta: -Entonces, ¿cuando el programa llega al Breakpoint podré saber qué valor tiene una variable determinada?

-Efectivamente **María**, y podrás saber el valor de aquellas que tú decidas. De este modo a través de los puntos de ruptura y de las ejecuciones paso a paso podrás descubrir dónde puede haber errores en tus programas. Conocer bien las herramientas que el depurador nos ofrece es algo que puede ahorrarnos mucho trabajo -Aporta **Ada**.



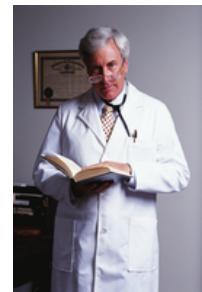
Ministerio de Educación y FP (CC BY-NC)

La fase de pruebas de una aplicación software es una etapa fundamental que en numerosas ocasiones puede suponer un tanto por ciento importante del tiempo de desarrollo del proyecto. Esta fase es planificada por los desarrolladores/analistas de acuerdo a los requisitos establecidos en la etapa de análisis, por lo tanto, no se trata de una fase improvisada. Dentro de las pruebas a realizar se encuentran las conocidas como .....pruebas de caja negra o .....pruebas de caja blanca.

Durante la etapa de implementación, los programadores deben realizar otros tipos de prueba que verifiquen el código que escriben.

La **Depuración de programas** es el proceso por el cual se identifican y corrigen errores de programación. Generalmente, en el argot de programación se utiliza la palabra debugging, que significa localización y eliminación de bichos (bugs) o errores de programa. A través de este proceso se descubren los errores y se identifica qué zonas del programa los producen. Hay tres etapas por las que un programa pasa cuando éste es desarrollado y que pueden generar errores:

- ✓ **Compilación:** Una vez que hemos terminado de afinar un programa, solemos pasar generalmente cierto tiempo eliminando errores de compilación. El compilador de Java mostrará una serie de chequeos en el código, sacando a la luz errores que pueden no apreciarse a simple vista. Una vez que el programa es liberado de los errores de compilación, obtendremos de él algunos resultados visibles pero quizás no haga aún lo que queremos.
- ✓ **Enlazado:** Todos los programas hacen uso de librerías de métodos y otros utilizan métodos generados por los propios programadores. Un método es enlazado (linked) sólo cuando éste es llamado, durante el proceso de ejecución. Pero cuando el programa es compilado, se realizan comprobaciones para saber si los métodos llamados existen y sus parámetros son correctos en número y tipo. Así que los errores de enlazado son detectados durante la etapa de compilación.
- ✓ **Ejecución:** Cuando el programa entra en ejecución, es muy frecuente que éste no funcione como se esperaba. De hecho, es normal que el programa falle. Algunos errores serán detectados automáticamente y el programador será informado, como acceder a una parte de un array que no existe (error de índices) por ejemplo. Otros son más sutiles y dan lugar simplemente a comportamientos no esperados, debido a la existencia de errores ocultos (bugs) en el programa. De ahí los términos bug y debugging. El problema de la depuración es que los síntomas de un posible error son generalmente poco claros, hay que recurrir a una labor de investigación para encontrar la causa.



Stockbyte CD-DVD Num. V07  
(CC BY-NC)

La depuración de programas es algo así como ser doctor: existe un síntoma, hemos de encontrar la causa y entonces determinar el problema. El trabajo de eliminación de errores puede ser interesante. La depuración y la prueba suelen requerir una cantidad de tiempo considerable en comparación con el tiempo dedicado a la primera codificación del programa. Pero no te preocupes, es normal emplear más tiempo en este proceso.

¿Y cómo llevamos a cabo la depuración de nuestros programas?, pues a través del debugger o depurador del sistema de desarrollo Java que estemos utilizando. Este depurador será una herramienta que nos ayudará a eliminar posibles errores de nuestro programa. Podremos utilizar depuradores simples, como el **jdb propio de Java** basado en línea de comandos. O bien, utilizar el **depurador existente en nuestro IDE** (en nuestro caso NetBeans). Este último tipo de depuradores muestra los siguientes elementos en pantalla: El programa en funcionamiento, el código fuente del programa y los nombres y valores actuales de las variables que se seleccionen.

¿Qué elementos podemos utilizar en el depurador? Existen al menos dos elementos fundamentales que podemos utilizar en nuestro debugger o depurador, son los siguientes:

- ✓ **Breakpoints o puntos de ruptura:** Estos puntos pueden ser determinados por el propio programador a lo largo del código fuente de su aplicación. Un Breakpoint es un lugar en el programa en el que la ejecución se detiene. Estos puntos se insertan en una determinada línea del código, entonces el programa se pone en funcionamiento y cuando el flujo de ejecución llega hasta él, la ejecución queda congelada y un puntero indica el lugar en el que la ejecución se ha detenido. El depurador muestra los valores de las variables tal y como están en ese momento de la ejecución. Cualquier discrepancia entre el valor actual y el valor que deberían tener supone una importante información para el proceso de depuración.
- ✓ **Ejecución paso a paso:** El depurador también nos permite ejecutar un programa paso a paso, es decir, línea por línea. A través de esta herramienta podremos seguir el progreso de ejecución de nuestra aplicación y supervisar su funcionamiento. Cuando la ejecución no es la esperada quizás estemos cerca de localizar un error o bug. En ocasiones, si utilizamos métodos procedentes de la biblioteca estándar no necesitaremos hacer un recorrido paso a paso por el interior de estos métodos, ya que es seguro que no contendrán errores internos y podremos ahorrar tiempo no entrando en su interior paso a paso. El debugger ofrece la posibilidad de entrar o no en dicho métodos.

## Debes conocer

Para completar tus conocimientos sobre la depuración de programas, te proponemos los siguientes enlaces en los que podrás encontrar cómo se llevan a cabo las tareas básicas de depuración a través del IDE NetBeans.

[Uso básico del depurador en NetBeans.](#)

## Para saber más

Si deseas conocer algo más sobre depuración de programas, pero a un nivel algo más avanzado, puedes ver el siguiente vídeo.

[Debugging avanzado en NetBeans.](#)

<https://www.youtube.com/embed/qAo9-bWDzj0>

[Resumen textual alternativo](#)

## Recomendación

La depuración de programas es una tarea fundamental en el desarrollo de software. Cuando escribimos código en cualquier lenguaje de programación, hay altas probabilidades de que genere errores durante la ejecución. **Cuando un error no se descubre de un vistazo rápido al código fuente, algo que suele ocurrir cuando el número de líneas de código aumenta, es totalmente recomendable utilizar el depurador.** Al principio cuesta hacerse con la rutina de depurar, pero es el método más efectivo para encontrar errores. **Ningún programador puede ser eficiente si no utiliza convenientemente el depurador de código.**

## 8.- Documentación del código.

### Caso práctico

**Ada** está mostrando a **Juan** la documentación sobre una serie de métodos estándar que van a necesitar para completar el desarrollo de una parte de la aplicación. Esta documentación tiene un formato estructurado y puede accederse a ella a través del navegador web.

-jQué útil y fácil está siendo el acceso a esta documentación! La verdad es que los que la han generado se lo han currado bastante. ¿Generar esta documentación se llevará mucho tiempo, no Ada? -Pregunta **Juan** mientras recoge de la impresora la documentación impresa.

**Ada** prepara rápidamente una clase en blanco y comienza a incorporarle una serie de comentarios: -Verás Juan, documentar el código es vital y si incorporas a tu código fuente unos comentarios en el formato que te voy a mostrar, la documentación puede ser generada automáticamente a través de la herramienta [Javadoc](#). Observa... -Responde **Ada**.



Dreftymac (CC BY-SA)

Llegados a este punto, vamos a considerar una cuestión de gran importancia, la documentación del código fuente. Piensa en las siguientes cuestiones:

- ✓ ¿Quién crees que accederá a la documentación del código fuente? Pues serán los autores del propio código u otros desarrolladores.
- ✓ ¿Por qué hemos de documentar nuestro código? Porque facilitaremos su mantenimiento y reutilización.
- ✓ ¿Qué debemos documentar? Obligatoriamente: clases, paquetes, constructores, métodos y atributos. Opcionalmente: bucles, partes de algoritmos que estimemos oportuno comentar, ...

A lo largo de nuestra vida como programadores es probable que nos veamos en la necesidad de reutilizar, modificar y mantener nuestro propio código o incluso, código de otros desarrolladores. ¿No crees que sería muy útil que dicho código estuviera convenientemente documentado? ¿Cuántas veces no hemos leído código de otros programadores y quizás no hayamos comprendido qué estaban haciendo en tal o cual método? Como podrás comprender, la generación de una documentación adecuada de nuestros programas puede suponer una inestimable ayuda para realizar ciertos procesos en el software.

Si analizamos la documentación de las clases proporcionada en los paquetes que distribuye Sun, nos daremos cuenta de que dicha documentación ha sido generada con una herramienta llamada [Javadoc](#). Pues bien, nosotros también podemos generar la documentación de nuestro código a través de dicha herramienta.

Si desde el principio nos acostumbramos a documentar el funcionamiento de nuestras clases desde el propio código fuente, estaremos facilitando la generación de la futura documentación de nuestras aplicaciones. ¿Cómo lo logramos? A través de una serie de comentarios especiales, llamados **comentarios de documentación** que serán tomados por [Javadoc](#) para generar una serie de archivos [HTML](#) que permitirán posteriormente, navegar por nuestra documentación con cualquier navegador web.

Los comentarios de documentación tienen una marca de comienzo (\*\*\*) y una marca de fin (\*). En su interior podremos encontrar dos partes diferenciadas: una para realizar una descripción y otra en la que encontraremos más etiquetas de documentación. Veamos un ejemplo:

```
/**  
 * Descripción principal (texto/HTML)  
 *  
 * Etiquetas (texto/HTML)  
 */
```

Este es el formato general de un comentario de documentación. Comenzamos con la marca de comienzo en una línea. Cada línea de comentario comenzará con un asterisco. El final del comentario de documentación deberá incorporar la

marca de fin. Las dos partes diferenciadas de este comentario son:

- ✓ **Zona de descripción:** es aquella en la que el programador escribe un comentario sobre la clase, atributo, constructor o método que se vaya a codificar bajo el comentario. Se puede incluir la cantidad de texto que se necesite, pudiendo añadir etiquetas HTML que formateen el texto escrito y así ofrecer una visualización mejorada al generar la documentación mediante Javadoc.
- ✓ **Zona de etiquetas:** en esta parte se colocará un conjunto de etiquetas de documentación a las que se asocian textos. Cada etiqueta tendrá un significado especial y aparecerán en lugares determinados de la documentación, una vez haya sido generada.

En la siguiente imagen puedes observar un ejemplo de un comentario de documentación.



José Luis García Martínez ([CC BY-NC](#))

## Citas para pensar

R. Caron: "do not document bad code - rewrite it".

## Reflexiona

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué. Documentar un programa no es sólo un acto de buen hacer del programador por aquello de dejar la obra rematada. Es además una necesidad que sólo se aprecia en su debida magnitud cuando hay errores que reparar o hay que extender el programa con nuevas capacidades o adaptarlo a un nuevo escenario.

[Mostrar retroalimentación](#)

Efectivamente esta afirmación es totalmente cierta, documentar es muy importante y hacerlo bien es una sana costumbre para todos los programadores y programadoras.

## 8.1.- Etiquetas y posición.

---

Cuando hemos de incorporar determinadas etiquetas a nuestros comentarios de documentación es necesario conocer dónde y qué etiquetas colocar, según el tipo de código que estemos documentando en ese momento. Existirán dos tipos generales de etiquetas:

1. **Etiquetas de bloque:** Son etiquetas que sólo pueden ser incluidas en el bloque de documentación, después de la descripción principal y comienzan con el símbolo @.
2. **Etiquetas en texto:** Son etiquetas que pueden ponerse en cualquier punto de la descripción o en cualquier punto de la documentación asociada a una etiqueta de bloque. Son etiquetas definidas entre llaves, de la siguiente forma  
{@etiqueta}

En la siguiente tabla podrás encontrar una referencia sobre las diferentes etiquetas y su ámbito de uso.

	@name	{@name}	{@description}	{@parameter}	{@example}	{@unit}	{@return}
Descripción	✓	✓	✓	✓	✓	✓	✓
Propiedad	✓	✓	✓	✓	✓	✓	✓
Clases e interfaces	✓	✓	✓	✓	✓	✓	✓
Attribución	✓	✓	✓	✓	✓	✓	✓
Comentarios de texto y bloques				✓	✓	✓	✓

José Luis García Martínez. ([CC BY-NC](#))

## 8.2.- Uso de las etiquetas.

¿Cuáles son las etiquetas típicas y su significado? Pasaremos seguidamente a enumerar y describir la función de las etiquetas más habituales a la hora de generar comentarios de documentación.

- ✓ **@autor texto con el nombre:** Esta etiqueta sólo se admite en clases e interfaces. El texto después de la etiqueta no necesitará un formato especial. Podremos incluir tantas etiquetas de este tipo como necesitemos.
- ✓ **@version texto de la versión:** El texto de la versión no necesitará un formato especial. Es conveniente incluir el número de la versión y la fecha de ésta. Podremos incluir varias etiquetas de este tipo una detrás de otra.
- ✓ **@deprecated texto:** Indica que no debería utilizarse, indicando en el texto las causas de ello. Se puede utilizar en todos los apartados de la documentación. Si se ha realizado una sustitución debería indicarse qué utilizar en su lugar. Por ejemplo:



Lensim (CC BY-SA)

```
@deprecated El método no funciona correctamente. Se recomienda el uso de {@link metodoCorrecto}
```

- ✓ **@exception nombre-excepción texto:** Esta etiqueta es equivalente a @throws.

**@param nombre-atributo texto:** Esta etiqueta es aplicable a parámetros de constructores y métodos. Describe los parámetros del constructor o método. Nombre-atributo es idéntico al nombre del parámetro. Cada etiqueta @param irá seguida del nombre del parámetro y después de una descripción de éste. Por ejemplo:

```
@param fromIndex: El índice del primer elemento que debe ser eliminado.
```

- ✓ **@exception nombre-excepción texto:** Esta etiqueta se puede omitir en los métodos que devuelven void. Deberá aparecer en todos los métodos, dejando explícito qué tipo o clase de valor devuelve y sus posibles rangos de valores. Veamos un ejemplo:

```
/**  
 * Chequea si un vector no contiene elementos.  
 *  
 * @return <code>verdadero</code>si solo si este vector no contiene componentes, esto es, su tamaño es cero;  
 * <code>falso</code> en cualquier otro caso.  
 */  
public boolean VectorVacio() {  
    return elementCount == 0;  
}
```

- ✓ **@see referencia:** Se aplica a clases, interfaces, constructores, métodos, atributos y paquetes. Añade enlaces de referencia a otras partes de la documentación. Podremos añadir a la etiqueta: cadenas de caracteres, enlaces HTML a páginas y a otras zonas del código. Por ejemplo:

```
* @see "Diseño de patrones: La reusabilidad de los elementos de la programación orientada a objetos"  
* @see <a href="http://www.w3.org/WAI/">Web Accessibility Initiative</a>  
* @see String#equals(Object) equals
```

- ✓ **@throws nombre-excepción texto:** En nombre-excepción tendremos que indicar el nombre completo de ésta. Podremos añadir una etiqueta por cada excepción que se lance explícitamente con una cláusula throws, pero siguiendo el orden alfabético. Esta etiquetas es aplicable a constructores y métodos, describiendo las posibles excepciones del constructor/método.

### Autoevaluación

¿Qué etiqueta podría omitirse en un método que devuelve `void`?

- `@param`.
- `@throws`.
- `@return`.

No es correcto, `@param` debe incluirse si existen parámetros en el método.

Incorrecto, `@throws` debe incluirse si el método lanza excepciones.

Correcto, ya que `@return` no sería necesaria al no devolver nada el método.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

## 8.3.- Orden de las etiquetas.

Las etiquetas deben disponerse en un orden determinado, ese orden es el siguiente:

### Orden de etiquetas de comentarios de documentación.

Etiqueta.	Descripción.
@autor	En clases e interfaces. Se pueden poner varios. Es mejor ponerlas en orden cronológico.
@version	En clases e interfaces.
@param	En métodos y constructores. Se colocarán tantos como parámetros tenga el constructor o método. Mejor en el mismo orden en el que se encuentren declarados.
@return	En métodos.
@exception	En constructores y métodos. Mejor en el mismo orden en el que se han declarado, o en orden alfabético.
@throws	Es equivalente a @exception.
@see	Podemos poner varios. Comenzaremos por los más generales y después los más específicos.
@deprecated	

### Para saber más

Si quieres conocer cómo obtener a través de Javadoc la documentación de tus aplicaciones, sigue los siguientes enlaces:

[Documentación con Javadoc.](#)

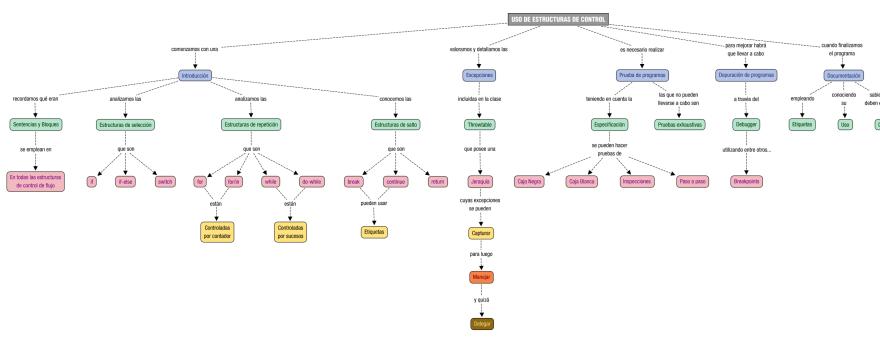
En el siguiente vídeo puedes observar cómo generar la documentación de un proyecto Java desde Netbeans.

<https://www.youtube.com/embed/KChdcRscFt0>

[Resumen textual alternativo](#)

## 9.- Conclusiones.

En esta unidad 4 hemos trabajado una parte importante del lenguaje Java: las estructuras de control de flujo, aquellas que nos permiten variar el flujo normal de ejecución dependiendo de ciertas condiciones. Hemos trabajado con estructuras condicionales, tanto simples como compuestas, así como con estructuras repetitivas, conocidas como bucles. Además, hemos introducido el concepto de Excepción, útil para controlar errores y darles tratamiento a los mismos. Por último, hemos hablado sobre la necesidad de documentar el código y hemos analizado las posibilidades de generar documentación automática a través de Netbeans. Por último, hemos introducido el depurador de código como una herramienta indispensable para cualquier programador, pues nos ayuda a detectar errores de ejecución de manera eficiente. En el siguiente mapa conceptual se pueden observar los conceptos que deben quedar claros antes de avanzar a la siguiente unidad:



Ministerio de Educación y FP (CC BY-NC)

En la siguiente unidad vamos a profundizar en el uso de clases y objetos, tanto en la definición de clases y objetos como en su creación y uso.



## Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

Tras su primer contacto con la **Programación Orientada a Objetos**, **María** está mucho más convencida de los beneficios que se pueden obtener utilizando este tipo de metodología. Las explicaciones de **Juan** y su breve experiencia con los **objetos** le hacen ahora ver el mundo de la programación desde otro punto de vista. Este nuevo método de trabajo les va a permitir distribuir las tareas de desarrollo de una manera más eficiente y poder adaptarse a cambios, correcciones y mejoras con mucho menos esfuerzo a la hora de modificar código.

**María** está convencida de que es el momento de empezar a escribir el código de sus propias **clases** para, a continuación, pasárselas a otros compañeros que las puedan usarlas. Una vez que otras personas reciban esas clases, podrán desentenderse de cómo están hechas por dentro (principios de **encapsulamiento** y **ocultación** de información) y limitarse únicamente a utilizarlas a través de sus métodos.

Para llevar a cabo esta labor **Juan** va a proseguir con las explicaciones con las que empezó cuando comenzaron a trabajar con objetos e introdujo el concepto de clase:

- Ha llegado el momento de empezar a desarrollar nuestras propias clases -le dice **Juan**.
  - Genial, vamos a comenzar a desarrollar la **primera biblioteca** de clases para **BK Programación**, ¿no es así?
  - Bueno, ésa es la idea. A ver si tenemos nuestro primer paquete de clases, que podríamos llamar `com.bkprogramacion`.
- Pues venga... ¡vamos allá!



[Ministerio de Educación y Formación Profesional. \(Dominio público\)](#)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

## 1.- Concepto de clase.

### Caso práctico



[Priority Pet Hospital \(CC BY-ND\)](#)

El equipo de trabajo que han formado **María y Juan** parece funcionar bastante bien. El proyecto que tienen entre manos para la aplicación informática de la clínica veterinaria empieza a tomar forma respecto a los requisitos que el cliente necesita. Es el momento de que empiecen a pensar en los distintos objetos con los que pueden modelar la información cuyo tratamiento desean automatizar. Muchos de estos objetos no van a ser proporcionados por las bibliotecas de Java, de manera que habrá que "fabricarlos", es decir, van a tener que diseñar e implementar las clases (los "moldes" o "plantillas") que les permitirán crear los objetos que van a necesitar. Ha llegado el momento de enfrentarse al **concepto de clase**.

Como ya has visto en anteriores unidades, las **clases** están compuestas por **atributos** y **métodos**. Una **clase** especifica las características comunes de un conjunto de **objetos**. De esta forma los programas que escribas estarán formados por un conjunto de **clases** a partir de las cuales irás creando **objetos** que se interrelacionarán unos con otros.

### Recomendación

En esta unidad se va a utilizar el concepto de **objeto** así como algunas de las diversas **estructuras de control** básicas que ofrece cualquier lenguaje de programación. Todos esos conceptos han sido explicados y utilizados en las unidades anteriores. Si consideras que es necesario hacer un repaso del concepto de objeto o del uso de las estructuras de control elementales, éste es el momento de hacerlo.

## 1.1- Repaso del concepto de objeto.

Desde el comienzo del módulo llevas utilizando el concepto de **objeto** para desarrollar tus programas de ejemplo. En las unidades anteriores se ha descrito un objeto como una entidad que contiene **información** y que es capaz de realizar ciertas **operaciones** con esa información. Según los valores que tenga esa información el objeto tendrá un **estado** determinado y según las operaciones que pueda llevar a cabo con esos datos será responsable de un **comportamiento** concreto.

Recuerda que entre las características fundamentales de un objeto se encontraban la **identidad** (los objetos son únicos y por tanto distinguibles entre sí, aunque pueda haber objetos exactamente iguales), un **estado** (los atributos que describen al objeto y los valores que tienen en cada momento) y un determinado **comportamiento** (acciones que se pueden realizar sobre el objeto).

Algunos ejemplos de objetos que podríamos imaginar podrían ser:

- ✓ Un coche de color rojo, marca SEAT, modelo Toledo, del año 2003. En este ejemplo tenemos una serie de atributos, como el color (en este caso rojo), la marca, el modelo, el año, etc. Así mismo también podríamos imaginar determinadas características como la cantidad de combustible que le queda, o el número de kilómetros recorridos hasta el momento.
- ✓ Un coche de color amarillo, marca Opel, modelo Astra, del año 2002.
- ✓ Otro coche de color amarillo, marca Opel, modelo Astra y también del año 2002. Se trataría de otro objeto con las mismas propiedades que el anterior, pero sería un segundo objeto.
- ✓ Un cocodrilo de cuatro metros de longitud y de veinte años de edad.
- ✓ Un círculo de radio 2 centímetros, con centro en las coordenadas (0,0) y relleno de color amarillo.
- ✓ Un círculo de radio 3 centímetros, con centro en las coordenadas (1,2) y relleno de color verde.

Si observas los ejemplos anteriores podrás distinguir sin demasiada dificultad al menos tres familias de objetos diferentes, que no tienen nada que ver una con otra:

- ✓ Los coches.
- ✓ Los círculos.
- ✓ Los cocodrilos.

Es de suponer entonces que cada objeto tendrá determinadas posibilidades de **comportamiento (acciones)** dependiendo de la familia a la que pertenezcan. Por ejemplo, en el caso de los **coches** podríamos imaginar acciones como: arrancar, frenar, acelerar, cambiar de marcha, etc. En el caso de los **cocodrilos** podrías imaginar otras acciones como: desplazarse, comer, dormir, cazar, etc. Para el caso del **círculo** se podrían plantear acciones como: cálculo de la superficie del círculo, cálculo de la longitud de la circunferencia que lo rodea, etc.



[Ministerio de Fomento \(CC BY-NC-ND\)](#)

Por otro lado, también podrías imaginar algunos **atributos** cuyos valores podrían ir cambiando en función de las acciones que se realizaran sobre el objeto: ubicación del coche (coordenadas), velocidad instantánea, kilómetros recorridos, velocidad media, cantidad de combustible en el depósito, etc. En el caso de los cocodrilos podrías imaginar otros atributos como: peso actual, el número de dientes actuales (irá perdiendo algunos a lo largo de su vida), el número de presas que ha cazado hasta el momento, etc.

Como puedes ver, un **objeto** puede ser cualquier cosa que puedas describir en términos de **atributos** y **acciones**.

Un objeto no es más que la representación de cualquier entidad concreta o abstracta que puedas percibir o imaginar y que pueda resultar de utilidad para modelar los elementos el entorno del problema que deseas resolver.

### Autoevaluación

Tenemos un objeto bombilla, de marca ACME, que se puede encender o apagar, que tiene una potencia de 50 vatios y ha costado 3 euros. La bombilla se encuentra en este momento apagada. A partir de esta información, ¿sabrías decir qué atributos y qué acciones (comportamiento) podríamos relacionar con ese objeto bombilla?

- Objeto bombilla con atributos potencia (50 vatios), precio (3 euros), marca (ACME) y estado (apagada).

Las acciones que se podrían ejercer sobre el objeto serían encender y apagar.

- Objeto bombilla con atributos precio (3 euros), marca (ACME) y apagado. Las acciones que se podrían ejercer sobre el objeto serían encender y apagar.
- Objeto bombilla con atributos precio (3 euros), marca (ACME), potencia (50 vatios) y estado (apagada). No se puede ejercer ninguna acción sobre el objeto.
- Se trata de un objeto bombilla cuyas posibles acciones son encender, apagar y arreglar. Sus atributos serían los mismos que en el caso a).

¡Así es! ¡Enhorabuena!

¡Error! En efecto la bombilla está apagada, pero eso sería un posible valor del atributo más que el nombre del atributo en sí mismo (que podríamos llamar por ejemplo estado y cuyo valor sería en este caso apagado). Además, faltaría el también atributo potencia (con valor de 50 vatios).

¡Has fallado! Los atributos son válidos, pero faltan las dos acciones que se han mencionado: encender y apagar.

¡No! En efecto los atributos son los del caso a), pero en el enunciado no se ha mencionado ninguna acción que sea arreglar la bombilla. Por supuesto que se trata una acción podría plantearse hacer con una bombilla real, pero en el contexto del problema (enunciado) no se ha mencionado nada al respecto de esa posible acción.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 1.2.- El concepto de clase.



Está claro que dentro de un mismo programa tendrás la oportunidad de encontrar decenas, cientos o incluso miles de objetos. En algunos casos no se parecerán en nada unos a otros, pero también podrás observar que habrá muchos que tengan un gran parecido, compartiendo un mismo comportamiento y unos mismos atributos. Habrá muchos objetos que sólo se diferenciaran por los valores que toman algunos de esos atributos.

Es aquí donde entra en escena el concepto de **clase**. Está claro que no podemos definir la estructura y el comportamiento de cada objeto cada vez que va a ser utilizado dentro de un programa, pues la escritura del código sería una tarea interminable y redundante. La idea es poder disponer de una **plantilla** o **modelo** para cada conjunto de objetos que sean del mismo tipo, es decir, que tengan los mismos atributos y un comportamiento similar.

watz (CC BY-NC-SA)

Una clase consiste en la definición de un tipo de objeto. Se trata de una descripción detallada de cómo van a ser los objetos que pertenezcan a esa clase indicando qué tipo de información contendrán (atributos) y cómo se podrá interactuar con ellos (comportamiento).

Como ya has visto en unidades anteriores, una clase consiste en un plantilla en la que se especifican:

- ✓ Los **atributos** que van a ser comunes a todos los objetos que pertenezcan a esa clase (información).
- ✓ Los **métodos** que permiten interactuar con esos objetos (comportamiento).

A partir de este momento podrás hablar ya sin confusión de objetos y de clases, sabiendo que los primeros son instancias concretas de las segundas, que no son más que una abstracción o definición.

Si nos volvemos a fijar en los ejemplos de objetos del apartado anterior podríamos observar que las clases serían lo que clasificamos como "familias" de objetos (coches, cocodrilos y círculos).

En el lenguaje cotidiano de muchos programadores puede ser habitual la confusión entre los términos clase y objeto. Aunque normalmente el contexto nos permite distinguir si nos estamos refiriendo realmente a una clase (definición abstracta) o a un objeto (instancia concreta), hay que tener cuidado con su uso para no dar lugar a interpretaciones erróneas, especialmente durante el proceso de aprendizaje.

### Autoevaluación

Un objeto y una clase en realidad hacen referencia al mismo concepto. Podría decirse que son sinónimos. ¿Verdadero o falso?

Verdadero  Falso

#### Falso

No es cierto. Aunque a veces ambos términos son utilizados para referirnos a lo mismo, no se trata del mismo concepto. Una clase consiste en una definición abstracta de cómo puede ser un objeto (qué información contiene y cómo se puede interactuar con él), pero no se refiere a ninguna instancia específica de objeto (podríamos decir que especificamos qué atributos puede tener, pero no qué valor tienen). Por otro lado un objeto es una instancia concreta de una clase, es decir, un conjunto de valores concretos para una definición genérica de clase.

## 2.- Estructura y miembros de una clase.

### Caso práctico



Ministerio de Educación (Elaboración propia). [\(CC BY-NC\)](#)

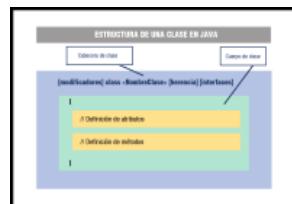
**María** empieza a tener bastante más claro en qué consiste una clase y cuál es la diferencia respecto a un objeto. Es el momento de empezar a crear en Java algunas de las clases que han estado pensando que podrían ser útiles para su aplicación. Para ello es necesario saber cómo se declara una clase en un lenguaje de programación determinado. Los becarios **Ana** y **Carlos** intuyen que van a comenzar a ver cómo está hecha una clase "por dentro". Parece ser que es el momento de empezar a tomar notas:

-De acuerdo, ya hemos diseñado algunas de las clases que queremos para nuestra aplicación. Pero, ¿cómo escribimos eso en Java? ¿Cómo declaramos o definimos una clase en Java? ¿Qué palabras reservadas hay que utilizar? ¿Qué partes tiene esa definición? -pregunta **María** con interés.

-Bien, es el momento de ver cómo es la estructura de una clase y cómo podemos escribirla en Java para luego poder fabricar objetos que pertenezcan a esa clase -le responde **Juan**.

En unidades anteriores ya se indicó que para declarar una clase en Java se usa la palabra reservada `class`. En la declaración de una clase vas a encontrar:

- ✓ **Cabecera de la clase.** Compuesta por una serie de modificadores de acceso, la palabra reservada `class` y el nombre de la clase.
- ✓ **Cuerpo de la clase.** En él se especifican los distintos miembros de la clase: **atributos** y **métodos**. Es decir, el contenido de la clase.



Ministerio de Educación [\(CC BY-NC\)](#)

Como puedes observar, el **cuerpo de la clase** es donde se declaran los **atributos** que caracterizan a los objetos de la clase y donde se define e implementa el comportamiento de dichos objetos; es decir, donde se declaran e implementan los **métodos**.

### Autoevaluación

Toda definición de una clase consta de cabecera y cuerpo. En la cabecera se definen los atributos de los objetos que se crearán a partir de esa clase y en el cuerpo estarán definidos los distintos métodos disponibles para manipular esos objetos. ¿Verdadero o falso?

- Verdadero  Falso

Falso

No es cierto. En la cabecera se define únicamente el nombre de la clase y algunos modificadores de acceso. En el cuerpo se definen tanto los atributos como los métodos.

## 2.1.- Declaración de una clase.

La declaración de una clase en Java tiene la siguiente estructura general:

```
[modificadores] class <NombreClase> [herencia] [interfaces] { // Cabecera de la clase  
    // Cuerpo de la clase  
  
    Declaración de los atributos  
  
    Declaración de los métodos  
  
}
```

Un ejemplo básico pero completo podría ser:

```
/**  
 *  
 * Ejemplo de clase Punto 2D  
 */  
class Punto {  
    // Atributos  
    int x, y;  
  
    // Métodos  
    int obtenerX () { return x; }  
    int obtenerY () { return y; }  
    void establecerX (int vx) { x = vx; }  
    void establecerY (int vy) { y = vy; }  
}
```

Diosdado Sánchez Hernández. ([CC BY-NC](#))

[Código de la clase Punto.](#) (1 KB)

En este caso se trata de una clase muy sencilla en la que el cuerpo de la clase (el área entre las llaves) contiene el código y las declaraciones necesarias para que los objetos que se construyan (basándose en esta clase) puedan funcionar apropiadamente en un programa (declaraciones de atributos para contener el estado del objeto y métodos que implementen el comportamiento de la clase y los objetos creados a partir de ella).

Si te fijas en los distintos programas que se han desarrollado en los ejemplos de las unidades anteriores, podrás observar que cada uno de esos programas era en sí mismo una clase Java: se declaraban con la palabra reservada `class` y contenían algunos atributos (variables) así como algunos métodos (como mínimo el método `main`).

En el ejemplo anterior hemos visto lo mínimo que se tiene que indicar en la **cabecera de una clase** (el nombre de la clase y la palabra reservada `class`). Se puede proporcionar bastante más información mediante modificadores y otros indicadores como por ejemplo el nombre de su **superclase** (si es que esa clase hereda de otra), si implementa algún **interfaz** y algunas cosas más que irás aprendiendo poco a poco.

A la hora de implementar una clase Java (escribirla en un archivo con un editor de textos o con alguna herramienta integrada como por ejemplo **Netbeans** o **Eclipse**) debes tener en cuenta:

- ✓ Por convenio, se ha decidido que en lenguaje Java los nombres de las clases deben de **empezar por una letra mayúscula**. Así, cada vez que observes en el código una palabra con la primera letra en mayúscula sabrás que se trata de una clase sin necesidad de tener que buscar su declaración. Además, **si el nombre de la clase está formado por varias palabras, cada una de ellas también tendrá su primera letra en mayúscula**. Siguiendo esta recomendación, algunos ejemplos de nombres de clases podrían ser: Recta, Circulo, Coche, CocheDeportivo, Jugador, JugadorFutbol, AnimalMarino, AnimalAcuatico, etc.
- ✓ El archivo en el que se encuentra una clase Java debe tener el mismo nombre que esa clase si queremos poder utilizarla desde otras clases que se encuentren fuera de ese archivo (**clase principal del archivo**).
- ✓ Tanto la definición como la implementación de una clase se incluye en el mismo archivo (archivo ".java"). En otros lenguajes como por ejemplo C++, definición e implementación podrían ir en archivos separados (por ejemplo en C++, serían sendos archivos con extensiones ".h" y ".cpp").

### Para saber más

Si quieres ampliar un poco más sobre este tema puedes echar un vistazo a los tutoriales de iniciación de Java en el sitio web de Oracle (en inglés):

[Java Classes.](#)



## 2.2.- Cabecera de una clase.

En general, la declaración de una clase puede incluir los siguientes elementos y en el siguiente orden:

1. Modificadores tales como `public`, `abstract` o `final`.
2. El nombre de la clase (con la primera letra de cada palabra en mayúsculas, por convenio).
3. El nombre de su **clase padre (superclase)**, si es que se especifica, precedido por la palabra reservada `extends` ("extiende" o "hereda de").
4. Una lista separada por comas de **interfaces** que son implementadas por la clase, precedida por la palabra reservada `implements` ("implementa").
5. El cuerpo de la clase, encerrado entre llaves {}.



Sleepy Valley (CC BY-NC-ND)

La sintaxis completa de una cabecera (los cuatro primeros puntos) queda de la forma:

[modificadores]

```
class <NombreClase> [<extends <NombreSuperClase>]<implements <NombreInterface1> [<<implements <NombreInterface2>] ...]
```

En el ejemplo anterior de la clase `Punto` teníamos la siguiente cabecera:

```
class Punto {
```

En este caso no hay **modificadores**, ni indicadores de **herencia**, ni implementación de **interfaces**. Tan solo la palabra reservada `class` y el nombre de la clase. Es lo mínimo que puede haber en la cabecera de una clase.

La **herencia** y las **interfaces** las verás más adelante. Vamos a ver ahora cuáles son los **modificadores** que se pueden indicar al crear la clase y qué efectos tienen. Los **modificadores de clase** son:

[`public`] [`final` | `abstract`]

Veamos qué significado tiene cada uno de ellos:

- ✓ **Modificador `public`.** Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase. Es decir, desde cualquier otra parte del programa. Si no se especifica este modificador, la clase sólo podrá ser utilizada desde clases que estén en el mismo **paquete**. El concepto de paquete lo veremos más adelante. Sólo puede haber una clase `public` (clase principal) en un archivo .java. El resto de clases que se definan en ese archivo no serán públicas.
- ✓ **Modificador `abstract`.** Indica que la clase es **abstracta**. Una clase abstracta no es instanciable. Es decir, no es posible crear objetos de esa clase (habrá que utilizar clases que hereden de ella). En este momento es posible que te parezca que no tenga sentido que esto pueda suceder (si no puedes crear objetos de esa clase, ¿para qué la quieres?), pero puede resultar útil a la hora de crear una jerarquía de clases. Esto lo verás también más adelante al estudiar el concepto de **herencia**.
- ✓ **Modificador `final`.** Indica que no podrás crear clases que hereden de ella. También volverás a este modificador cuando estudies el concepto de **herencia**. Los modificadores `final` y `abstract` son excluyentes (sólo se puede utilizar uno de ellos).

Todos estos modificadores y palabras reservadas las iremos viendo poco a poco, así que no te preocupes demasiado por intentar entender todas ellas en este momento.

En el ejemplo anterior de la clase `Punto` tendríamos una clase que sería sólo visible (utilizable) desde el mismo paquete en el que se encuentra la clase (modificador de acceso por omisión o de paquete, o `package`). Desde fuera de ese paquete no sería visible o accesible. Para poder utilizarla desde cualquier parte del código del programa bastaría con añadir el atributo `public`: `public class Punto`.

### Autoevaluación

**Si queremos poder instanciar objetos de una clase desde cualquier parte de un programa, ¿qué modificador o modificadores habrá que utilizar en su declaración?**

- `private`.
- `public`.
- `abstract`.
- Ninguno de los anteriores.

¡Incorrecto! El modificador `private` no puede utilizarse en la cabecera de una clase.

¡Correcto! En efecto, el modificador `public` es el que permite que una clase sea accesible desde cualquier paquete.

¡Has fallado! El modificador `abstract` es para indicar que la clase es abstracta.

¡Error! Alguno de los anteriores sí es correcto.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 2.3.- Cuerpo de una clase.

Como ya has visto anteriormente, el cuerpo de una clase se encuentra encerrado entre llaves y contiene la declaración e implementación de sus miembros. Los miembros de una clase pueden ser:

- ✓ **Atributos**, que especifican los datos que podrá contener un objeto de la clase.
- ✓ **Métodos**, que implementan las acciones que se podrán realizar con un objeto de la clase.



[Sleepy Valley \(CC BY-NC-ND\)](#)

Una clase puede no contener en su declaración atributos o métodos, pero debe de contener al menos uno de los dos (la clase no puede ser vacía).

En el ejemplo anterior donde se definía una clase `Punto`, tendríamos los siguientes atributos:

- ✓ Atributo `x`, de tipo `int`.
- ✓ Atributo `y`, de tipo `int`.

Eso decir, dos valores de tipo entero. Cualquier objeto de la clase `Punto` que sea creado almacenará en su interior dos números enteros (`x` e `y`). Cada objeto diferente de la clase `Punto` contendrá sendos valores `x` e `y`, que podrán coincidir o no con el contenido de otros objetos de esa misma clase `Punto`.

Por ejemplo, si se han declarado varios objetos de tipo `Punto`:

`Punto p1, p2, p3;`

Sabremos que cada uno de esos objetos `p1`, `p2` y `p3` contendrán un par de coordenadas (`x, y`) que definen el estado de ese objeto. Puede que esos valores coincidan con los de otros objetos de tipo `Punto`, o puede que no, pero en cualquier caso serán objetos diferentes creados a partir del mismo molde (de la misma clase).

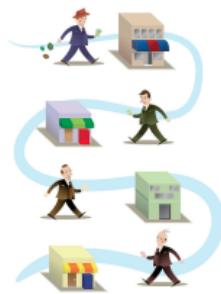
Por otro lado, la clase `Punto` también definía una serie de métodos:

```
int obtenerX () { return x; }

int obtenerY() { return y; }

void establecerX (int vx) { x= vx; }

void establecerY (int vy) { y= vy; };
```



[Sleepy Valley \(CC BY-NC-ND\)](#)

Cada uno de esos métodos puede ser llamado desde cualquier objeto que sea una instancia de la clase `Punto`. Se trata de operaciones que permiten manipular los datos (atributos) contenidos en el objeto bien para calcular otros datos o bien para modificar los propios atributos.

### Autoevaluación

**Si disponemos de varios objetos que han sido creados a partir de la misma definición de clase, en realidad tendremos un único objeto, pues hacen referencia a un mismo tipo de clase (plantilla). ¿Verdadero o falso?**

- Verdadero  Falso

**Falso**

No es cierto. El hecho de que varios objetos hayan sido creados a partir de la misma definición no significa en absoluto que se trate el mismo objeto. Cada uno de ellos es un objeto diferente y con existencia propia de manera independiente de los demás. Por ejemplo, la creación de tres objetos basados en la clase bombilla no significa que los tres objetos sean el mismo. Cada uno será una bombilla diferente, un objeto distinto, aunque los tres hayan sido creados basándose en el mismo modelo (el de

una bombilla). Puede incluso que los tres tengan los mismos valores en sus atributos y se trate de tres bombillas iguales, pero serán tres bombillas.

## 2.4.- Miembros estáticos o de clase.

---

Cada vez que se produce una instancia de una clase (es decir, se crea un objeto de esa clase), se desencadenan una serie de procesos (construcción del objeto) que dan lugar a la creación en memoria de un espacio físico que constituirá el objeto creado. De esta manera cada objeto tendrá sus propios miembros a imagen y semejanza de la plantilla propuesta por la clase.

Por otro lado, podrás encontrarte con ocasiones en las que determinados miembros de la clase (atributos o métodos) no tienen demasiado sentido como partes del objeto, sino más bien como partes de la clase en sí (partes de la plantilla, pero no de cada instancia de esa plantilla). Por ejemplo, si creamos una clase Coche y quisieramos disponer de un atributo con el nombre de la clase (un atributo de tipo `String` con la cadena "Coche"), no tiene mucho sentido replicar ese atributo para todos los objetos de la clase Coche, pues para todos va a tener siempre el mismo valor (la cadena "Coche"). Es más, ese atributo puede tener sentido y existencia al margen de la existencia de cualquier objeto de tipo Coche. Podría no haberse creado ningún objeto de la clase Coche y sin embargo seguiría teniendo sentido poder acceder a ese atributo de nombre de la clase, pues se trata en efecto de un atributo de la propia clase más que de un atributo de cada objeto instancia de la clase.

Para poder definir miembros estáticos en Java se utiliza el modificador `static`. Los miembros (tanto atributos como métodos) declarados utilizando este modificador son conocidos como miembros estáticos o miembros de clase. A continuación vas a estudiar la creación y utilización de atributos y métodos. En cada caso verás cómo declarar y usar atributos estáticos y métodos estáticos.

## 3.- Atributos.

### Caso práctico



Ministerio de Educación (CC BY-NC)

**María** está entusiasmada con las posibilidades que le ofrece el concepto de clase para poder crear cualquier tipo de objeto que a ella se le ocurra. Ya ha aprendido cómo declarar la cabecera de una clase en Java y ha estado probando con algunos de los ejemplos que le ha proporcionado **Juan**. Entiende más o menos cómo funcionan algunos de los modificadores de la clase, pero ha visto que el cuerpo es algo más complejo: ya no se trata de una simple línea de código como en el caso de la cabecera. Ahora tiene un conjunto de líneas de código que parecen hasta cierto punto un programa en pequeño. Puede encontrarse con declaraciones de variables, estructuras de control, realización de cálculos, etc.

Lo primero que **María** ha observado es que al principio suele haber algunas declaraciones de variables:

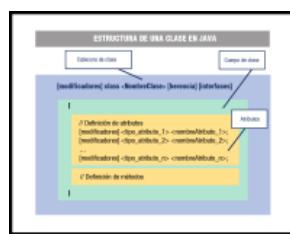
-¿Estas declaraciones de variables son lo que hemos llamado atributos?-le pregunta a **Juan**.

-Así es -contesta **Ada**, que en ese momento acaba de entrar por la puerta con **Ana** y con **Carlos**.

-Ahora que estás todos juntos, creo que ha llegado el momento que os explique algunas cosas acerca de los miembros de una clase. Vamos a empezar por los atributos.

Los **atributos** constituyen la estructura interna de los objetos de una clase. Se trata del conjunto de datos que los objetos de una determinada clase almacenan cuando son creados. Es decir es como si fueran variables cuyo ámbito de existencia es el objeto dentro del cual han sido creadas. Fuera del objeto esas variables no tienen sentido y si el objeto deja de existir, esas variables también deberían hacerlo (proceso de destrucción del objeto). Los atributos a veces también son conocidos con el nombre de **variables miembro** o **variables de objeto**.

Los atributos pueden ser de cualquier tipo de los que pueda ser cualquier otra variable en un programa en Java: desde tipos elementales como **int**, **boolean** o **float** hasta tipos referenciados como **arrays**, **Strings** u **objetos**.



Ministerio de Educación (CC BY-NC)

Además del tipo y del nombre, la declaración de un atributo puede contener también algunos modificadores (como por ejemplo **public**, **private**, **protected** o **static**). Por ejemplo, en el caso de la clase **Punto** que habíamos definido en el apartado anterior podrías haber declarado sus atributos como:

```
public int x;  
public int y;
```

De esta manera estarías indicando que ambos atributos son públicos, es decir, accesibles por cualquier parte del código programa que tenga acceso a un objeto de esa clase.

Como ya verás más adelante al estudiar el concepto de **encapsulación**, lo normal es declarar todos los atributos (o al menos la mayoría) como privados (**private**) de manera que si se desea acceder o manipular algún atributo se tenga que hacer a través de los métodos proporcionados por la clase.

## Autoevaluación

Dado que normalmente se pretende encapsular el contenido de un objeto en su interior y permitir el acceso a sus atributos únicamente a través de los métodos, los atributos de una clase suelen declararse con el modificador `public`. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

No es cierto. Precisamente porque se pretende ocultar los atributos al código externo a una clase, éstos no deberían de ser públicos sino privados. Es decir, que si queremos que el acceso a un atributo pueda hacerse únicamente a través de los métodos de la clase, los atributos deberán ser privados (modificador `private`).

### 3.1.- Declaración de atributos.

---

La sintaxis general para la declaración de un atributo en el interior de una clase es:

```
[modificadores] <tipo> <nombreAtributo>;
```

Ejemplos:

```
int x;  
public int elementoX, elementoY;  
private int x, y, z;  
static double descuentoGeneral;  
final bool casado;
```

Te suena bastante, ¿verdad? La declaración de los atributos en una clase es exactamente igual a la declaración de cualquier variable tal y como has estudiado en las unidades anteriores y similar a como se hace en cualquier lenguaje de programación. Es decir mediante la indicación del tipo y a continuación el nombre del atributo, pudiéndose declarar varios atributos del mismo tipo mediante una lista de nombres de atributos separada por comas (exactamente como ya has estudiado al declarar variables).

La declaración de un **atributo** (o **variable miembro** o **variable de objeto**) consiste en la declaración de una variable que únicamente existe en el interior del objeto y por tanto su vida comenzará cuando el objeto comience a existir (el objeto sea creado). Esto significa que cada vez que se cree un objeto se crearán tantas variables como atributos contenga ese objeto en su interior (definidas en la clase, que es la plantilla o "molde" del objeto). Todas esas variables estarán encapsuladas dentro del objeto y sólo tendrán sentido dentro de él.

En el ejemplo que estamos utilizando de objetos de tipo **Punto** (instancias de la clase **Punto**), cada vez que se cree un nuevo Punto **p1**, se crearán sendos atributos **x**, **y** de tipo **int** que estarán en el interior de ese punto **p1**. Si a continuación se crea un nuevo objeto **Punto p2**, se crearán otros dos nuevos atributos **x**, **y** de tipo **int** que estarán esta vez alojados en el interior de **p2**. Y así sucesivamente...

Dentro de la declaración de un atributo puedes encontrar tres partes:

- ✓ **Modificadores.** Son palabras reservadas que permiten modificar la utilización del atributo (indicar el control de acceso, si el atributo es constante, si se trata de un atributo de clase, etc.). Los iremos viendo uno a uno.
- ✓ **Tipo.** Indica el tipo del atributo. Puede tratarse de un tipo primitivo (**int**, **char**, **bool**, **double**, etc) o bien de uno referenciado (objeto, array, etc.).
- ✓ **Nombre.** Identificador único para el nombre del atributo. Por convenio se **suelen utilizar las minúsculas**. En caso de que se trate de un identificador que contenga varias palabras, **a partir de la segunda palabra se suele poner la letra de cada palabra en mayúsculas**. Por ejemplo: **primerValor**, **valor**, **puertalzquierda**, **cuartoTrasero**, **equipoVecendor**, **sumaTotal**, **nombreCandidatoFinal**, etc. Cualquier identificador válido de Java será admitido como nombre de atributo válido, pero es importante seguir este convenio para facilitar la legibilidad del código (todos los programadores de Java lo utilizan).

Como puedes observar, los atributos de una clase también pueden contener modificadores en su declaración (como sucedía al declarar la propia clase). Estos modificadores permiten indicar cierto comportamiento de un atributo a la hora de utilizarlo. Entre los modificadores de un atributo podemos distinguir:

- ✓ **Modificadores de acceso.** Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí. Sólo se puede poner uno.
- ✓ **Modificadores de contenido.** No son excluyentes. Pueden aparecer varios a la vez.
- ✓ **Otros modificadores:** **transient** y **volatile**. El primero se utiliza para indicar que un atributo es transitorio (no persistente) y el segundo es para indicar al compilador que no debe realizar optimizaciones sobre esa variable. Es más que probable que no necesites utilizarlos en este módulo.

Aquí tienes la sintaxis completa de la declaración de un atributo teniendo en cuenta la lista de todos los modificadores e indicando cuáles son incompatibles unos con otros:

```
[private | protected | public] [static] [final] [transient] [volatile] <tipo> <nombreAtributo>;
```

Vamos a estudiar con detalle cada uno de ellos.

## 3.2.- Modificadores de acceso.

Los modificadores de acceso disponibles en Java para un atributo son:

- ✓ Modificador de acceso **por omisión** (o **de paquete**). Si no se indica ningún modificador de acceso en la declaración del atributo, se utilizará este tipo de acceso. Se permitirá el acceso a este atributo desde todas las clases que estén dentro del **mismo paquete (package)** que esta clase (la que contiene el atributo que se está declarando). No es necesario escribir ninguna palabra reservada. Si no se pone nada se supone se desea indicar este modo de acceso.
- ✓ Modificador de acceso **public**. Indica que **cualquier clase** (por muy ajena o lejana que sea) tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (**public**).
- ✓ Modificador de acceso **private**. Indica que sólo se puede acceder al atributo desde **dentro de la propia clase**. El atributo estará "oculto" para cualquier otra zona de código fuera de la clase en la que está declarado el atributo. Es lo opuesto a lo que permite **public**.
- ✓ Modificador de acceso **protected**. En este caso se permitirá acceder al atributo desde cualquier **subclase** (lo verás más adelante al estudiar la **herencia**) de la clase en la que se encuentre declarado el atributo, y también desde las clases del **mismo paquete**.



Ministerio de Educación (CC BY-NC)

A continuación puedes observar un resumen de los distintos niveles accesibilidad que permite cada modificador:

**Cuadro de niveles accesibilidad a los atributos de una clase.**

	Misma clase	Subclase	Mismo paquete	Otro paquete
<b>Sin modificador (paquete)</b>	Sí		Sí	
<b>public</b>	Sí	Sí	Sí	Sí
<b>private</b>	Sí			
<b>protected</b>	Sí	Sí	Sí	

¡Recuerda que los modificadores de acceso son excluyentes! Sólo se puede utilizar uno de ellos en la declaración de un atributo.

### Ejercicio resuelto

Imagina que quieres escribir una clase que represente un **rectángulo** en el plano. Para ello has pensado en los siguientes atributos:

- ✓ Atributos **x1, y1**, que representan la coordenadas del vértice inferior izquierdo del rectángulo. Ambos de tipo **double** (números reales).
- ✓ Atributos **x2, y2**, que representan las coordenadas del vértice superior derecho del rectángulo. También de tipo **double** (números reales).

Con estos dos puntos ( $x_1, y_1$ ) y ( $x_2, y_2$ ) se puede definir perfectamente la ubicación de un rectángulo en el plano.

Escribe una clase que contenga todos esos atributos teniendo en cuenta que queremos que sea una clase visible desde cualquier parte del programa y que sus atributos sean también accesibles desde cualquier parte del código.

[Mostrar retroalimentación](#)

Dado que se trata de una clase que podrá usarse desde cualquier parte del programa, utilizaremos el modificador de acceso **public** para la clase:

```
public class Rectangulo
```

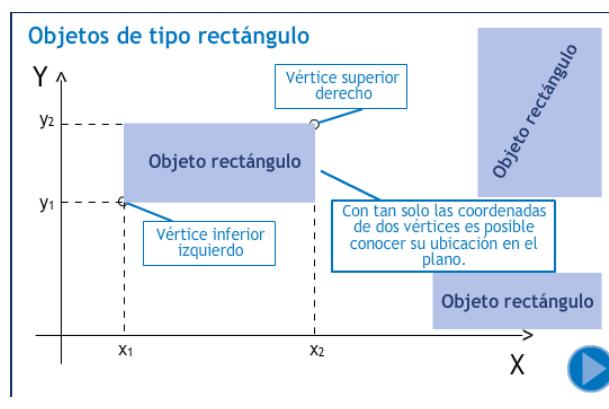
Los cuatro atributos que necesitamos también han de ser visibles desde cualquier parte, así que también se utilizará el modificador de acceso `public` para los atributos:

```
public double x1, y1; // Vértice inferior izquierdo  
public double x2, y2; // Vértice superior derecho
```

De esta manera la clase completa quedaría:

```
public class Rectangulo {  
  
    public double x1, y1; // Vértice inferior izquierdo  
  
    public double x2, y2; // Vértice superior derecho  
  
}
```

Como muestra la siguiente imagen, un objeto Rectángulo contiene los datos necesarios para representarlo en un plano.



## Recomendación

Para conseguir las ventajas de la encapsulación, se recomienda que los atributos de una clase se declaren **privados o tengan acceso de paquete (por omisión)**.

Para acceder a los atributos de una clase desde el exterior se deben proporcionar métodos que sean públicos. Eso garantizará que los atributos no sean modificados directamente desde el exterior. Los más conocidos son los métodos getters y setters. Los conoceremos mas adelante.

### 3.3.- Modificadores de contenido.

Los modificadores de contenido **no son excluyentes** (pueden aparecer varios para un mismo atributo). Son los siguientes:

- ✓ **Modificador static.** Hace que el atributo sea común para todos los objetos de una misma clase. Es decir, todas las clases compartirán ese mismo atributo con el mismo valor. Es un caso de miembro estático o miembro de clase: un **atributo estático** o **atributo de clase** o **variable de clase**.
- ✓ **Modificador final.** Indica que el atributo es una **constante**. Su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los **atributos constantes** (**final**) se escribe con **todas las letras en mayúsculas**.

[jorel314 \(CC BY\)](#)

En el siguiente apartado sobre atributos estáticos verás un ejemplo completo de un atributo estático (**static**). Veamos ahora un ejemplo de atributo constante (**final**).

Imagina que estás diseñando un conjunto de clases para trabajar con expresiones geométricas (figuras, superficies, volúmenes, etc.) y necesitas utilizar muy a menudo la constante pi con abundantes cifras significativas, por ejemplo, 3.14159265. Utilizar esa constante literal muy a menudo puede resultar tedioso además de poco operativo (imagina que el futuro hubiera que cambiar la cantidad de cifras significativas). La idea es declararla una sola vez, asociarle un nombre simbólico (un identificador) y utilizar ese identificador cada vez que se necesite la constante. En tal caso puede resultar muy útil declarar un atributo **final** con el valor 3.14159265 dentro de la clase en la que se considere oportuno utilizarla. El mejor identificador que podrías utilizar para ella será probablemente el propio nombre de la constante (y en mayúsculas, para seguir el convenio de nombres), es decir, **PI**.

Así podría quedar la declaración del atributo:

```
class claseGeometria {  
    // Declaración de constantes  
    public final float PI= 3.14159265;  
    ...
```

### Autoevaluación

¿Con qué modificador puede indicarse en Java que un atributo es constante?

- Con el modificador **constant**.
- Con el modificador **starter**.
- Con el modificador **final**.
- Con el modificador **static**.

¡Error! Ese modificador no existe en Java.

¡Has fallado! Es modificador no existe en Java.

¡Así es! ¡Enhorabuena! El modificador **final** es el que se utiliza para declarar un atributo como constante.

¡No! ¡Error! El modificador **static** se utiliza para declarar miembros estáticos o de clase.

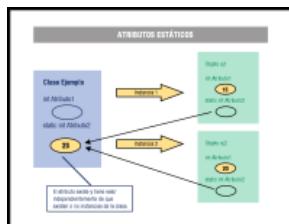
### Solución

3.14159265358979323846243838279  
0793164062862089986280369253421170627  
823 06647 09344  
823 06647 09344  
46 09550 58223  
17 23359 4081  
28478 1117  
4502 8410  
2701 9385  
21205 5544  
46229 48954  
9303 81984  
4298 81984  
66593 34461  
284756 48233  
78678 31652 71  
3234601 66610454326648 86  
9234603 66610454326648  
2133936 0726024914127  
132092296 132092296

- 1. Incorrecto
- 2. Incorrecto
- 3. Opción correcta
- 4. Incorrecto

## 3.4.- Atributos estáticos.

Como ya has visto, el modificador `static` hace que el atributo sea común (el mismo) para todos los objetos de una misma clase. En este caso sí podría decirse que la existencia del atributo no depende de la existencia del objeto, sino de la propia clase y por tanto sólo habrá uno, independientemente del número de objetos que se creen. El atributo será siempre el mismo para todos los objetos y tendrá un valor único independientemente de cada objeto. Es más, aunque no exista ningún objeto de esa clase, el atributo sí existirá y podrá contener un valor (pues se trata de un **atributo de la clase** más que del objeto).



Ministerio de Educación (CC BY-NC)

Uno de los ejemplos más habituales (y sencillos) de atributos estáticos o de clase es el de un **contador** que indica el número de objetos de esa clase que se han ido creando. Por ejemplo, en la clase de ejemplo `Punto` podrías incluir un atributo que fuera ese contador para llevar un registro del número de objetos de la clase `Punto` que se van construyendo durante la ejecución del programa.

Otro ejemplo de atributo estático (y en este caso también constante) que también se ha mencionado anteriormente al hablar de miembros estáticos era disponer de un atributo **nombre**, que contuviera un `String` con el nombre de la clase. Nuevamente ese atributo sólo tiene sentido para la clase, pues habrá de ser compartido por todos los objetos que sean de esa clase (es el nombre de la clase a la que pertenecen los objetos y por tanto siempre será la misma e igual para todos, no tiene sentido que cada objeto de tipo `Punto` almacene en su interior el nombre de la clase, eso lo debería hacer la propia clase).

```
class Punto {  
    // Coordenadas del punto  
  
    private int x, y;  
  
    // Atributos de clase: cantidad de puntos creados hasta el momento  
  
    public static cantidadPuntos;  
  
    public static final nombre;
```

Obviamente, para que esto funcione como estás pensando, también habrá que escribir el código necesario para que cada vez que se cree un objeto de la clase `Punto` se incremente el valor del atributo `cantidadPuntos`. Volverás a este ejemplo para implementar esa otra parte cuando estudies los constructores.

### Ejercicio resuelto

Ampliar el ejercicio anterior del rectángulo incluyendo los siguientes atributos:

- ✓ Atributo `numRectangulos`, que almacena el número de objetos de tipo rectángulo creados hasta el momento.
- ✓ Atributo `nombre`, que almacena el nombre que se le quiera dar a cada rectángulo.
- ✓ Atributo `nombreFigura`, que almacena el nombre de la clase, es decir, "Rectángulo".
- ✓ Atributo `PI`, que contiene el nombre de la constante PI con una precisión de cuatro cifras decimales.

No se desea que los atributos `nombre` y `numRectangulos` puedan ser visibles desde fuera de la clase. Y además se desea que la clase sea accesible solamente desde su propio paquete.

[Mostrar retroalimentación](#)

Los atributos **numRectangulos**, **nombreFigura** y **PI** podrían ser **estáticos** pues se trata de valores más asociados a la propia clase que a cada uno de los objetos que se puedan ir creando. Además, en el caso de **PI** y **nombreFigura**, también podría ser un **atributo final**, pues se trata de valores únicos y constantes (3.1416 en el caso de **PI** y "Rectángulo" en el caso de **nombreFigura**).

Dado que no se desea que se tenga accesibilidad a los atributos **nombre** y **numRectangulos** desde fuera de la clase podría utilizarse el atributo **private** para cada uno de ellos.

Por último hay que tener en cuenta que se desea que la clase sólo sea accesible desde el interior del paquete al que pertenece, por tanto habrá que utilizar el modificador por omisión o de paquete. Esto es, no incluir ningún modificador de acceso en la cabecera de la clase.

Teniendo en cuenta todo lo anterior la clase quedaría finalmente así:

```
class Rectangulo { // Sin modificador "public" para que sólo sea accesible desde el paquete

    // Atributos de clase

    private static int numRectangulos; // Número total de rectángulos creados

    public static final String nombreFigura= "Rectángulo"; // Nombre de la clase

    public static final double PI= 3.1416; // Constante PI

    // Atributos de objeto

    private String nombre; // Nombre del rectángulo

    public double x1, y1; // Vértice inferior izquierdo

    public double x2, y2; // Vértice superior derecho

}
```

## 4.- Métodos.

### Caso práctico



Ministerio de Educación (CC BY-NC)

**María** ya ha estado utilizando **métodos** para poder manipular algunos de los objetos que han creado en programas básicos de prueba. En el proyecto de la **Clínica Veterinaria** en el que está trabajando junto con **Juan** van a tener que crear bastantes tipos de objetos (clases) que representen el sistema de información que quieren modelar y automatizar. Ya han pensado y definido muchos de los atributos que van a tener esas clases. Ahora necesitan empezar a definir qué tipos de acciones se van a poder realizar sobre la información que contenga cada clase o familia de objetos:

-Ya tengo pensadas algunas de las acciones que van a ser necesarias para manipular algunas de las clases que hemos planteado -Le dice **María** a **Juan**.

-Muy bien. Entonces es el momento de empezar a definir métodos.

-Perfecto. ¿Y cómo lo hacemos? Cuando hemos utilizado objetos de clases ya incorporadas en el lenguaje, simplemente he utilizado sus métodos, pero aún no he declarado ninguno.

-No te preocupes, vamos a ver algunos ejemplos de declaración, implementación y utilización de métodos de una clase. Verás como es mucho más sencillo de lo que piensas.

Como ya has visto anteriormente, los **métodos** son las herramientas que nos sirven para definir el comportamiento de un objeto en sus interacciones con otros objetos. Forman parte de la estructura interna del objeto junto con los atributos.

En el proceso de declaración de una clase que estás estudiando ya has visto cómo escribir la cabecera de la clase y cómo especificar sus atributos dentro del cuerpo de la clase. Tan solo falta ya declarar los métodos, que estarán también en el interior del cuerpo de la clase junto con los atributos.

Los métodos suelen declararse después de los atributos. Aunque atributos y métodos pueden aparecer mezclados por todo el interior del cuerpo de la clase es aconsejable no hacerlo para mejorar la **claridad** y la **legibilidad** del código. De ese modo, cuando echemos un vistazo rápido al contenido de una clase, podremos ver rápidamente los atributos al principio (normalmente ocuparán menos líneas de código y serán fáciles de reconocer) y cada uno de los métodos inmediatamente después. Cada método puede ocupar un número de líneas de código más o menos grande en función de la complejidad del proceso que pretenda implementar.

Los métodos representan la **interfaz** de una clase. Son la forma que tienen otros objetos de comunicarse con un objeto determinado solicitándole cierta información o pidiéndole que lleve a cabo una determinada acción. Este modo de programar, como ya has visto en unidades anteriores, facilita mucho la tarea al desarrollador de aplicaciones, pues le permite abstraerse del contenido de las clases haciendo uso únicamente del interfaz (métodos).

### Autoevaluación

¿Qué elementos forman la interfaz de un objeto?

- Los atributos del objeto.
- Las variables locales de los métodos del objeto.
- Los métodos.
- Los atributos estáticos de la clase.

¡Error! Los atributos no deben ser la vía de comunicación de un objeto con otros objetos.

¡No! Las variables locales son objetos internos dentro de un método.

¡Así es! ¡Enhorabuena! Los métodos representan la interfaz de un objeto. Son el medio a través del cual otros objetos pueden comunicarse con ella.

¡Has fallado! Los atributos estáticos no deben ser la vía de comunicación de un objeto con otros objetos. No tiene mucho sentido pues los miembros estáticos pertenecen más a la clase que al objeto.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

## 4.1.- Declaración de un método.

La definición de un método se compone de dos partes:

- ✓ **Cabecera del método**, que contiene el nombre del método junto con el tipo devuelto, un conjunto de posibles modificadores y una lista de parámetros.
- ✓ **Cuerpo del método**, que contiene las sentencias que implementan el comportamiento del método (incluidas posibles sentencias de declaración de variables locales).

Los **elementos mínimos** que deben aparecer en la declaración de un método son:

- ✓ El tipo devuelto por el método.
- ✓ El nombre del método.
- ✓ Los paréntesis.
- ✓ El cuerpo del método entre llaves: {}.

Por ejemplo, en la clase **Punto** que se ha estado utilizando en los apartados anteriores podrías encontrar el siguiente método:

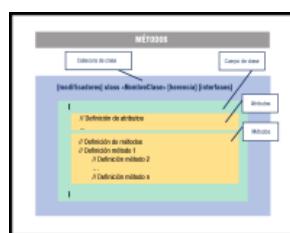
```
int obtenerX ()  
{  
    // Cuerpo del método  
    ...  
}
```

Donde:

- ✓ El tipo devuelto por el método es **int**.
- ✓ El nombre del método es **obtenerX**.
- ✓ No recibe ningún parámetro: aparece una lista vacía entre paréntesis: **0**.
- ✓ El cuerpo del método es todo el código que habría encerrado entre llaves: **{}.**

Dentro del cuerpo del método podrás encontrar declaraciones de variables, sentencias y todo tipo de estructuras de control (bucles, condiciones, etc.) que has estudiado en los apartados anteriores.

Ahora bien, la declaración de un método puede incluir algunos elementos más. Vamos a estudiar con detalle cada uno de ellos.



Ministerio de Educación ([CC BY-NC](#))

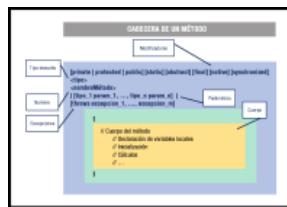
## 4.2.- Cabecera de método.

La declaración de un método puede incluir los siguientes elementos:

1. **Modificadores** (como por ejemplo los ya vistos `public` o `private`, más algunos otros que irás conociendo poco a poco). No es obligatorio incluir modificadores en la declaración.
2. El **tipo devuelto** (o tipo de retorno), que consiste en el tipo de dato (primitivo o referencia) que el método devuelve tras ser ejecutado. Si eliges `void` como tipo devuelto, el método no devolverá ningún valor.
3. El **nombre del método**, aplicándose para los nombres el mismo convenio que para los atributos.
4. Una **lista de parámetros** separados por comas y entre paréntesis donde cada parámetro debe ir precedido por su tipo. Si el método no tiene parámetros la lista estará vacía y únicamente aparecerán los paréntesis.
5. Una **lista de excepciones** que el método puede lanzar. Se utiliza la palabra reservada `throws` seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un método incluya una lista de excepciones, aunque muchas veces será conveniente. En unidades anteriores ya has trabajado con el concepto de excepción y más adelante volverás a hacer uso de ellas.
6. El **cuerpo del método**, encerrado entre llaves. El cuerpo contendrá el código del método (una lista sentencias y estructuras de control en lenguaje Java) así como la posible declaración de variables locales.

La sintaxis general de la cabecera de un método podría entonces quedar así:

```
[private | protected | public] [static] [abstract] [final] [native] [synchronized]  
<tipo> <nombreMétodo> ( [<lista_parametros>] )  
[throws <lista_excepciones>]
```



Ministerio de Educación (CC BY-NC)

Como sucede con todos los identificadores en Java (variables, clases, objetos, métodos, etc.), puede usarse cualquier identificador que cumpla las normas. Ahora bien, para mejorar la legibilidad del código, se ha establecido el siguiente convenio para nombrar los métodos: **utilizar un verbo en minúscula o bien un nombre formado por varias palabras que comience por un verbo en minúscula, seguido por adjetivos, nombres, etc. los cuales sí aparecerán en mayúsculas**.

Algunos ejemplos de métodos que siguen este convenio podrían ser: ejecutar, romper, mover, subir, responder, obtenerX, establecerValor, estaVacio, estaLleno, moverFicha, subirPalanca, responderRapido, girarRuedaIzquierda, abrirPuertaDelantera, CambiarMarcha, etc.

En el ejemplo de la clase `Punto`, puedes observar cómo los métodos `obtenerX` y `obtenerY` siguen el convenio de nombres para los métodos, devuelven en ambos casos un tipo `int`, su lista de parámetros es vacía (no tienen parámetros) y no lanzan ningún tipo de excepción:

```
int obtenerX ()  
int obtenerY ()
```

### Autoevaluación

¿Con cuál de los siguientes modificadores no puede ser declarado un método en Java?

- `private`.
- `extern`.

static.

public.

¡Error! El modificador `private` sí es un modificador válido para un método Java.

¡Así es! ¡Enhorabuena! El modificador `extern` no es aplicable para métodos. De hecho no es un modificador ni una palabra reservada en Java.

¡Error! El modificador `static` sí es un modificador válido para un método Java.

¡Has fallado! El modificador `public` sí es un modificador válido para un método Java.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 4.3.- Modificadores en la declaración de un método.

En la declaración de un método también pueden aparecer modificadores (como en la declaración de la clase o de los atributos). Un método puede tener los siguientes tipos de modificadores:

- ✓ **Modificadores de acceso.** Son los mismos que en el caso de los atributos (por omisión o de paquete, `public`, `private` y `protected`) y tienen el mismo cometido (acceso al método sólo por parte de clases del mismo paquete, o por cualquier parte del programa, o sólo para la propia clase, o también para las subclases).
- ✓ **Modificadores de contenido.** Son también los mismos que en el caso de los atributos (`static` y `final`) junto con, aunque su significado no es el mismo.
- ✓ **Otros modificadores** (no son aplicables a los atributos, sólo a los métodos): `abstract`, `native`, `synchronized`.

Un método **static** es un método cuya implementación es igual para todos los objetos de la clase y sólo tendrá acceso a los atributos estáticos de la clase (dado que se trata de un método de clase y no de objeto, sólo podrá acceder a la información de clase y no la de un objeto en particular). Este tipo de métodos pueden ser llamados sin necesidad de tener un objeto de la clase instanciado.

En Java un ejemplo típico de métodos estáticos se encuentra en la clase `Math`, cuyos métodos son todos estáticos (`Math.abs`, `Math.sin`, `Math.cos`, etc.). Como habrás podido comprobar en este ejemplo, la llamada a métodos estáticos se hace normalmente usando el nombre de la propia clase y no el de una instancia (objeto), pues se trata realmente de un método de clase. En cualquier caso, los objetos también admiten la invocación de los métodos estáticos de su clase y funcionaría correctamente.

Un método **final** es un método que no permite ser sobreescrito por las clases descendientes de la clase a la que pertenece el método. Volverás a ver este modificador cuando estudies en detalle la **herencia**.

El modificador **native** es utilizado para señalar que un método ha sido implementado en código nativo (en un lenguaje que ha sido compilado a lenguaje máquina, como por ejemplo **C** o **C++**). En estos casos simplemente se indica la cabecera del método, pues no tiene cuerpo escrito en Java.

Un método **abstract** (**método abstracto**) es un método que no tiene implementación (el cuerpo está vacío). La implementación será realizada en las clases descendientes. Un método sólo puede ser declarado como `abstract` si se encuentra dentro de una clase `abstract`. También volverás a este modificador en unidades posteriores cuando trabajes con la **herencia**.

Por último, si un método ha sido declarado como `synchronized`, el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine. Puede resultar útil si sabes que un determinado método va a poder ser llamado concurrentemente por varios procesos a la vez. Por ahora no lo vas a necesitar.

Dada la cantidad de modificadores que has visto hasta el momento y su posible aplicación en la declaración de clases, atributos o métodos, veamos un resumen de todos los que has visto y en qué casos pueden aplicarse:

**Cuadro de aplicabilidad de los modificadores.**

	Clase	Atributo	Método
<b>Sin modificador (paquete)</b>	Sí	Sí	Sí
<code>public</code>	Sí	Sí	Sí
<code>private</code>		Sí	Sí
<code>protected</code>	Sí	Sí	Sí
<code>static</code>		Sí	Sí
<code>final</code>	Sí	Sí	Sí
<code>synchronized</code>			Sí
<code>native</code>			Sí
<code>abstract</code>	Sí		Sí

## 4.4.- Parámetros en un método.

La lista de parámetros de un método se coloca tras el nombre del método. Esta lista estará constituida por pares de la forma "<tipoParametro> <nombreParametro>". Cada uno de esos pares estará separado por comas y la lista completa estará encerrada entre paréntesis:

```
<tipo> <nombreMetodo> ( <tipo_1> <nombreParametro_1>, <tipo_2> <nombreParametro_2>, ..., <tipo_r>
```



Si la lista de parámetros es vacía, tan solo aparecerán los paréntesis:

```
<tipo> <nombreMetodo> ( )
```

A la hora de declarar un método, debes tener en cuenta:

- ✓ Puedes incluir cualquier cantidad de parámetros. Se trata de una decisión del programador, pudiendo ser incluso una lista vacía.
- ✓ Los parámetros podrán ser de cualquier tipo (tipos primitivos, referencias, objetos, arrays, etc.).
- ✓ No está permitido que el nombre de una variable local del método coincida con el nombre de un parámetro.
- ✓ No puede haber dos parámetros con el mismo nombre. Se produciría ambigüedad.
- ✓ Si el nombre de algún parámetro coincide con el nombre de un atributo de la clase, éste será ocultado por el parámetro. Es decir, al indicar ese nombre en el código del método estarás haciendo referencia al parámetro y no al atributo. Para poder acceder al atributo tendrás que hacer uso del operador de autorreferencia `this`, que verás un poco más adelante.
- ✓ En Java el paso de parámetros es siempre por valor, excepto en el caso de los tipos referenciados (por ejemplo los objetos) en cuyo caso se está pasando efectivamente una referencia. La referencia (el objeto en sí mismo) no podrá ser cambiada pero sí elementos de su interior (atributos) a través de sus métodos o por acceso directo si se trata de un miembro público.

Es posible utilizar una construcción especial llamada `varargs` (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: "...") después del tipo del cual se puede tener una lista variable de argumentos, un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

```
<tipo> <nombreMetodo> (<tipo>... <nombre>)
```

Es posible además mezclar el uso de `varargs` con parámetros fijos. En tal caso, la lista de parámetros variables debe aparecer al final (y sólo puede aparecer una).

En realidad se trata una manera transparente de pasar un array con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que ir recorriendo el array para ir obteniendo cada uno de los elementos de la lista de argumentos variables.

### Para saber más

Si quieres ver algunos ejemplos de cómo utilizar el mecanismo de argumentos variables, puedes echar un vistazo a los siguientes enlaces (en inglés):

[VarArgs en Java.](#)

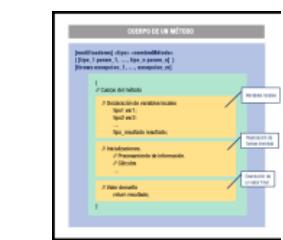
También puedes echar un vistazo al artículo general sobre paso de parámetros a métodos en los manuales de Oracle sobre Java:

[Passing Information to a Method or a Constructor.](#)

## 4.5.- Cuerpo de un método.

```
int obtenerX ()  
{  
    return x;  
}
```

El interior de un método (cuerpo) está compuesto por una serie de sentencias en lenguaje Java:



Ministerio de Educación ([CC BY-NC](#))

- ✓ Sentencias de **declaración de variables locales** al método.
- ✓ Sentencias que implementan la **lógica del método** (estructuras de control como bucles o condiciones; utilización de métodos de otros objetos; cálculo de expresiones matemáticas, lógicas o de cadenas; creación de nuevos objetos, etc.). Es decir, todo lo que has visto en las unidades anteriores.
- ✓ Sentencia de **devolución del valor de retorno** (`return`). Aparecerá al final del método y es la que permite devolver la información que se le ha pedido al método. Es la última parte del proceso y la forma de comunicarse con la parte de código que llamó al método (paso de mensaje de vuelta). Esta sentencia de devolución siempre tiene que aparecer al final del método. Tan solo si el tipo devuelto por el método es `void` (vacío) no debe aparecer (pues no hay que devolver nada al código llamante).

En el ejemplo de la clase `Punto`, tenías los métodos `obtenerX` y `obtenerY`. Veamos uno de ellos:

En ambos casos lo único que hace el método es precisamente devolver un valor (utilización de la sentencia `return`). No recibe parámetros (mensajes o información de entrada) ni hace cálculos, ni obtiene resultados intermedios o finales. Tan solo devuelve el contenido de un atributo. Se trata de uno de los métodos más sencillos que se pueden implementar: un método que devuelve el valor de un atributo. En inglés se les suele llamar métodos de tipo **get**, que en inglés significa **obtener**.

Además de esos dos métodos, la clase también disponía de otros dos que sirven para la función opuesta (`establecerX` y `establecerY`). Veamos uno de ellos:

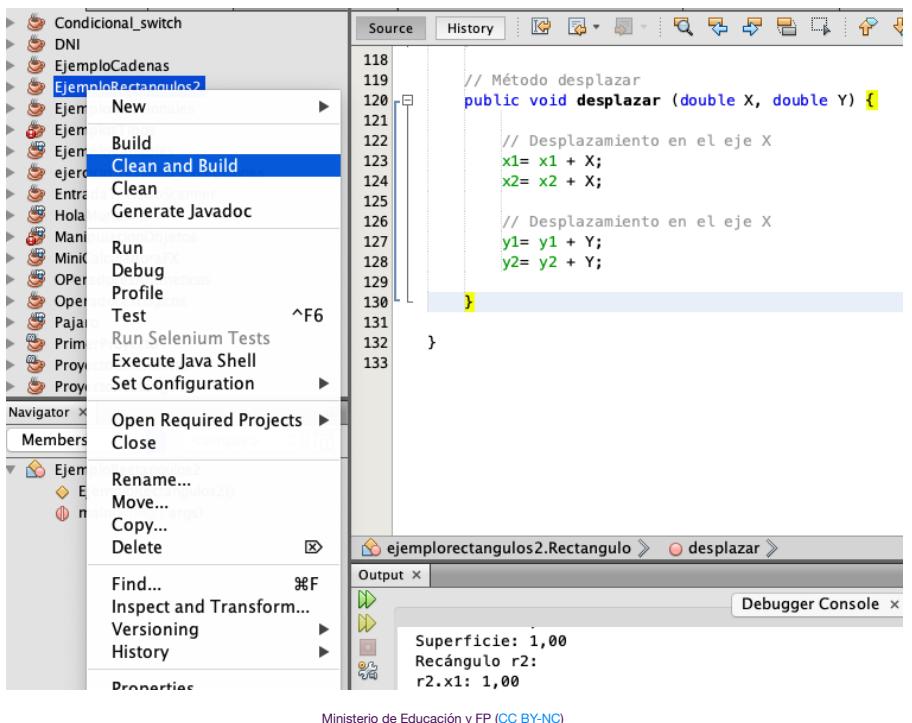
```
void establecerX (int vx)  
{  
    x= vx;  
}
```

En este caso se trata de pasar un valor al método (parámetro `vx` de tipo `int`) el cual será utilizado para modificar el contenido del atributo `x` del objeto. Como habrás podido comprobar, ahora no se devuelve ningún valor (el tipo devuelto es `void` y no hay sentencia `return`). En inglés se suele hablar de métodos de tipo **set**, que en inglés significa poner o fijar (**establecer** un valor). El método `establecerY` es prácticamente igual pero para establecer el valor del atributo `y`.

Normalmente el código en el interior de un método será algo más complejo y estará formado un conjunto de sentencias en las que se realizarán cálculos, se tomarán decisiones, se repetirán acciones, etc. Puedes ver un ejemplo más completo en el siguiente ejercicio.

### Debes conocer

Como ya comentamos en la primera unidad de trabajo y has ido comprobando a lo largo de las demás, Netbeans se encarga de compilar el código que escribimos en cuanto guardamos. Internamente va generando los ficheros `.class` necesarios para lanzar a ejecución las aplicaciones. Si en algún momento encontramos algún problema con la compilación automática, por ejemplo, porque detectamos que no se han actualizado convenientemente los ficheros `.class` y el resultado de ejecución es incorrecto, **podemos forzar la compilación de todo el proyecto**. Para ello, utilizamos la opción **Clean and Build**, accesible desde el menú contextual que aparece al hacer click con el botón derecho sobre un proyecto.



## Ejercicio resuelto

Vamos a seguir ampliando la clase en la que se representa un rectángulo en el plano (clase **Rectangulo**). Para ello has pensado en los siguientes métodos **públicos**:

- ✓ Métodos **obtenerNombre** y **establecerNombre**, que permiten el acceso y modificación del atributo **nombre** del rectángulo.
- ✓ Método **calcularSuperficie**, que calcula el área encerrada por el rectángulo.
- ✓ Método **calcularPerímetro**, que calcula la longitud del perímetro del rectángulo.
- ✓ Método **desplazar**, que mueve la ubicación del rectángulo en el plano en una cantidad X (para el eje X) y otra cantidad Y (para el eje Y). Se trata simplemente de sumar el desplazamiento X a las coordenadas x1 y x2, y el desplazamiento Y a las coordenadas y1 e y2. Los **parámetros** de entrada de este método serán por tanto X e Y, de tipo **double**.
- ✓ Método **obtenerNumRectangulos**, que devuelve el número de rectángulos creados hasta el momento.

Incluye la implementación de cada uno de esos métodos en la clase **Rectangulo**.

**¡Es muy recomendable que trates de implementarlo en Netbeans!**

**Si tienes dudas, revisa la solución.**

[Mostrar retroalimentación](#)

En el caso del método **obtenerNombre**, se trata simplemente de devolver el valor del atributo **nombre**:

```
public String obtenerNombre () {
    return nombre;
}
```

Para el implementar el método **establecerNombre** también es muy sencillo. Se trata de modificar el contenido del atributo **nombre** por el valor proporcionado a través de un parámetro de entrada:

```

public void establecerNombre (String nom) {
    nombre= nom;
}

```

Los métodos de cálculo de superficie y perímetro no van a recibir ningún parámetro de entrada, tan solo deben realizar cálculos a partir de los atributos contenidos en el objeto para obtener los resultados perseguidos. En cada caso habrá que aplicar la expresión matemática apropiada:

- ✓ En el caso de la superficie, habrá que calcular la longitud de la **base** y la **altura** del rectángulo a partir de las coordenadas de las esquinas inferior izquierda ( $x_1, y_1$ ) y superior derecha ( $x_2, y_2$ ) de la figura. La base sería la diferencia entre  $x_2$  y  $x_1$ , y la altura la diferencia entre  $y_2$  e  $y_1$ . A continuación tan solo tendrías que utilizar la consabida fórmula de "base por altura", es decir, una multiplicación.
- ✓ En el caso del perímetro habrá también que calcular la longitud de la **base** y de la **altura** del rectángulo y a continuación sumar dos veces la longitud de la base y dos veces la longitud de la altura.

En ambos casos el resultado final tendrá que ser devuelto a través de la sentencia `return`. También es aconsejable en ambos casos la utilización de variables locales para almacenar los cálculos intermedios (como la base o la altura).

```

public double calcularSuperficie () {
    double area, base, altura; // Variables locales

    // Cálculo de la base
    base= x2-x1;

    // Cálculo de la altura
    altura= y2-y1;

    // Cálculo del área
    area= base * altura;

    // Devolución del valor de retorno
    return area;
}

public double calcularPerimetro () {
    double perimetro, base, altura; // Variables locales

    // Cálculo de la base
    base= x2-x1;

    // Cálculo de la altura
    altura= y2-y1;

    // Cálculo del perímetro
    perimetro= 2*base + 2*altura;

    // Devolución del valor de retorno
    return perimetro;
}

```

En el caso del método **desplazar**, se trata de modificar:

- ✓ Los contenidos de los atributos  $x_1$  y  $x_2$  sumándoles el parámetro X,
- ✓ Los contenidos de los atributos  $y_1$  e  $y_2$  sumándoles el parámetro Y.

```
public void desplazar (double X, double Y) {  
    // Desplazamiento en el eje X  
    x1= x1 + X;  
    x2= x2 + X;  
    // Desplazamiento en el eje Y  
    y1= y1 + Y;  
    y2= y2 + Y;  
}
```

En este caso no se devuelve ningún valor (tipo devuelto vacío: `void`).

Por último, el método **obtenerNumRectangulos** simplemente debe devolver el valor del atributo **numRectangulos**. En este caso es razonable plantearse que este método podría ser más bien un método de clase (estático) más que un método de objeto, pues en realidad es una característica de la clase más que algún objeto en particular. Para ello tan solo tendrías que utilizar el modificador de acceso `static`:

```
public static int obtenerNumRectangulos () {  
    return numRectangulos;  
}
```

Para acceder al archivo completo de la clase Java puedes utilizar el siguiente enlace. Ten en cuenta que descargarás un proyecto con un fichero fuente `.java` que contiene la implementación de la clase Rectángulo. **¿Cómo puedes probar la clase Rectángulo?**

1. Añádele un **main** a la propia clase.
2. Crea una nueva clase que contenga un método **main**.

En el método **main** tendrás que instanciar objetos de la clase **Rectángulo** e invocar sus métodos. Esa será la forma de comprobar el funcionamiento de la clase. **Todo ésto lo tendrás mas claro cuando revises los contenidos del punto 6**. Puedes esperar hasta ese momento para realizar las pruebas.

[Clase Rectangulo.](#) (2.00 KB)

## 4.6.- Sobrecarga de métodos.

En principio podrías pensar que un método puede aparecer una sola vez en la declaración de una clase (no se debería repetir el mismo nombre para varios métodos). Pero no tiene porqué siempre suceder así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la **sobrecarga de métodos**.

El lenguaje Java soporta la característica conocida como **sobrecarga de métodos**. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.



Collin Key, (CC BY-SA)

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (`int`), un número real (`double`) o una cadena de caracteres (`String`). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo. Por ejemplo:

- ✓ Método `pintarEntero (int entero)`.
- ✓ Método `pintarReal (double real)`.
- ✓ Método `pintarCadena (double String)`.
- ✓ Método `pintarEnteroCadena (int entero, String cadena)`.

Y así sucesivamente para todos los casos que deseas contemplar...

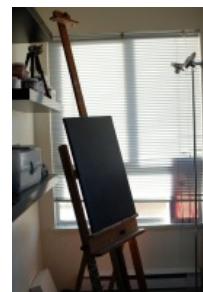
La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: `pintar`). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo `pintar`, para todos los métodos anteriores:

- ✓ Método `pintar (int entero)`.
- ✓ Método `pintar (double real)`.
- ✓ Método `pintar (double String)`.
- ✓ Método `pintar (int entero, String cadena)`.

En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro método `pintar (int entero)`, pues es imposible distinguirlo de otro método con el mismo nombre y con la misma lista de parámetros (ya existe un método `pintar` con un único parámetro de tipo `int`).

También debes tener en cuenta que el **tipo devuelto** por el método no es considerado a la hora de identificar un método, así que un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, **no podrías definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente**. El compilador producirá un error de duplicidad en el nombre del método y no te lo permitirá.



Svacher (CC BY-NC-ND)

Es conveniente no abusar de sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), dado que podría hacer el código menos legible.

### Autoevaluación

En una clase Java puedes definir tantos métodos con el mismo nombre como deseas y sin ningún tipo de restricción pues el lenguaje soporta la sobrecarga de métodos y el compilador sabrá distinguir unos métodos de otros. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

No es cierto del todo. La sobrecarga de métodos en efecto permite declarar distintos métodos con un mismo nombre, pero para que el compilador pueda distinguirlos es necesario que los métodos que tengan el mismo nombre tengan una lista de parámetros diferentes (en número de parámetros o en el tipo de los parámetros). Si no es así, será imposible distinguir unos métodos de otros y se producirá un error durante el proceso de compilación.

## 4.7.- Sobrecarga de operadores.

+ x + = +      Del mismo modo que hemos visto la posibilidad de sobrecargar métodos (disponer de varias versiones de un método con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobrecargar operadores del lenguaje tales como +, -, \*, ( ), <, >, etc. para darles otro significado dependiendo del tipo de objetos con los que vaya a operar.

- x - = +      En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

+ : + = +  
- : - = +  
+ : - = -  
- : + = -      Un típico ejemplo podría ser el de la sobrecarga de operadores aritméticos como la suma (+) o el producto (\*) para operar con fracciones. Si se definen objetos de una clase **Fracción** (que contendrá los atributos numerador y denominador) podrían sobrecargarse los operadores aritméticos (habría que redefinir el operador suma (+) para la suma, el operador asterisco (\*) para el producto, etc.) para esta clase y así podrían utilizarse para sumar o multiplicar objetos de tipo **Fracción** mediante el algoritmo específico de suma o de producto del objeto **Fracción** (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

Idite (CC BY-NC-SA)  
En algunos lenguajes de programación como por ejemplo C++ o C# se permite la sobrecarga, pero no es algo soportado en todos los lenguajes. ¿Qué sucede en el caso concreto de Java?

### El lenguaje Java no soporta la sobrecarga de operadores.

En el ejemplo anterior de los objetos de tipo **Fracción**, habrá que declarar métodos en la clase **Fracción** que se encarguen de realizar esas operaciones, pero no lo podremos hacer sobrecargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
public Fraccion sumar (Fraccion sumando)  
public Fraccion multiplicar (Fraccion multiplicando)
```

Y así sucesivamente...

Dado que en este módulo se está utilizando el lenguaje Java para aprender a programar, no podremos hacer uso de esta funcionalidad. Más adelante, cuando aprendas a programar en otros lenguajes, es posible que sí tengas la posibilidad de utilizar este recurso.

### Autoevaluación

La sobrecarga de operadores en Java permite “rescribir” el significado de operadores del lenguaje tales como +, -, \*, <, >, etc. Esto puede resultar muy útil a la hora de mejorar la legibilidad del código cuando definimos por ejemplo nuevos objetos matemáticos (números racionales, números complejos, conjuntos, etc.). ¿Verdadero o falso?

Verdadero  Falso

**Falso**

No es cierto. El lenguaje Java no soporta la sobrecarga de operadores.

## 4.8.- La referencia this.



Diosdado Sánchez Hernández ([CC BY-NC](#))

La palabra reservada `this` consiste en una referencia al objeto actual. El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre). En tales casos el parámetro "oculta" al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). En estos casos la referencia `this` nos permite acceder a estos atributos ocultados por los parámetros.

Dado que `this` es una referencia a la propia clase en la que te encuentras en ese momento, puedes acceder a sus atributos mediante el operador punto `(.)` como sucede con cualquier otra clase u objeto. Por tanto, en lugar de poner el nombre del atributo (que estos casos haría referencia al parámetro), podrías escribir `this.nombreAtributo`, de manera que el compilador sabrá que te estás refiriendo al atributo y se eliminará la ambigüedad.

En el ejemplo de la clase `Punto`, podríamos utilizar la referencia `this` si el nombre del parámetro del método coincidiera con el del atributo que se desea modificar. Por ejemplo

```
void establecerX (int x)
{
    this.x= x;
}
```

En este caso ha sido indispensable el uso de `this`, pues si no sería imposible saber en qué casos te estás refiriendo al parámetro `x` y en cuáles al atributo `x`. Para el compilador el identificador `x` será siempre el parámetro, pues ha "ocultado" al atributo.

En algunos casos puede resultar útil hacer uso de la referencia `this` aunque no sea necesario, pues puede ayudar a mejorar la legibilidad del código.

### Para saber más

Puedes echar un vistazo al artículo general sobre la referencia `this` en los manuales de Oracle (en inglés):

[Using the this Keyword.](#)

### Autoevaluación

**La referencia `this` en Java resulta muy útil cuando se quieren utilizar en un método nombres de parámetros que coinciden con los nombres de variables locales del método. ¿Verdadero o falso?**

Verdadero  Falso

#### Falso

En un método no puede coincidir el nombre de una variable local con el de un parámetro. Produciría un error de compilación, pues se estarían declarando dos variables con el mismo nombre (los parámetros pueden verse en cierto modo como variables locales del método). La utilidad del operador `this` es la de facilitar el uso de un parámetro con el mismo nombre que un atributo, pues aunque el atributo quede "oculto" por el parámetro, se podrá acceder al atributo gracias al operador `this (this.nombreAtributo)`, que es una referencia al propio objeto.

## Ejercicio resuelto

Modificar el método **obtenerNombre** de la clase **Rectangulo** de ejercicios anteriores utilizando la referencia **this**.

[Mostrar retroalimentación](#)

Si utilizamos la referencia **this** en este método, entonces podremos utilizar como identificador del parámetro el mismo identificador que tiene el atributo (aunque no tiene porqué hacerse si no se desea):

```
public void establecerNombre (String nombre) {  
    this.nombre= nombre;  
}
```

## 4.9.- Métodos estáticos.

Como ya has visto en ocasiones anteriores, un **método estático** es un método que puede ser usado directamente desde la clase, sin necesidad de tener que crear una instancia para poder utilizar al método. También son conocidos como **métodos de clase** (como sucedía con los **atributos de clase**), frente a los **métodos de objeto** (es necesario un objeto para poder disponer de ellos).

Los métodos estáticos no pueden manipular atributos de instancias (objetos) sino atributos estáticos (de clase) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la clase, más que para una instancia concreta.

Algunos ejemplos de operaciones que suelen realizarse desde métodos estáticos:



Ministerio de Educación y FP [\(CC BY-NC-SA\)](#)

- ✓ Acceso a atributos específicos de clase: incremento o decremento de contadores internos de la clase (no de instancias), acceso a un posible atributo de nombre de la clase, etc.
- ✓ Operaciones genéricas relacionadas con la clase pero que no utilizan atributos de instancia. Por ejemplo una clase **NIF** (o **DNI**) que permite trabajar con el DNI y la letra del NIF y que proporciona funciones adicionales para calcular la letra NIF de un número de DNI que se le pase como parámetro. Ese método puede ser interesante para ser usado desde fuera de la clase de manera independiente a la existencia de objetos de tipo NIF.

En la biblioteca de Java es muy habitual encontrarse con clases que proporcionan métodos estáticos que pueden resultar muy útiles para cálculos auxiliares, conversiones de tipos, etc. Por ejemplo, la mayoría de las clases del paquete `java.lang` que representan tipos (`Integer`, `String`, `Float`, `Double`, `Boolean`, etc.) ofrecen métodos estáticos para hacer conversiones. Aquí tienes algunos ejemplos:

- ✓ `static String valueOf (int i)`. Devuelve la representación en formato `String` (cadena) de un valor `int`. Se trata de un método que no tiene que ver nada en absoluto con instancias de concretas de `String`, sino de un método auxiliar que puede servir como herramienta para ser usada desde otras clases. Se utilizaría directamente con el nombre de la clase. Por ejemplo:

```
String enteroCadena= String.valueOf (23);
```

- ✓ `static String valueOf (float f)`. Algo similar para un valor de tipo `float`. Ejemplo de uso:

```
String floatCadena= String.valueOf (24.341);
```

- ✓ `static int parseInt (String s)`. En este caso se trata de un método estático de la clase `Integer`. Analiza la cadena pasada como parámetro y la transforma en un `int`. Ejemplo de uso:

```
int cadenaEntero= Integer.parseInt ("-12");
```

Todos los ejemplos anteriores son casos en los que se utiliza directamente la clase como una especie de **caja de herramientas** que contiene métodos que pueden ser utilizados desde cualquier parte, por eso suelen ser métodos públicos.

### Para saber más

Puedes echar un vistazo a algunas clases del paquete `java.lang` (por ejemplo `Integer`, `String`, `Float`, `Double`, `Boolean` y `Math`) y observar la gran cantidad de métodos estáticos que ofrecen para ser utilizados sin necesidad de tener que crear objetos de esas clases:

[Package.java.lang.](#)

## 5.- Encapsulación, control de acceso y visibilidad.

### Caso práctico



Ministerio de Educación [\(CC BY-NC\)](#)

Juan está desarrollando algunas de las clases que van a necesitar para el proyecto de la **Clínica Veterinaria** y empiezan a asaltarle dudas acerca de cuándo deben ser visibles unos u otros miembros. Recuerda ha estado viendo con María los distintos modificadores de acceso aplicables a las clases, atributos y métodos. Está claro que hasta que no empiece a utilizarlos para casos concretos y con aplicación práctica real no terminará de comprender exactamente su mecánica de funcionamiento: cuándo interesa ocultar un determinado miembro, cuándo interesa que otro miembro sea visible, en qué casos vale la pena crear un método para acceder al valor de un atributo, etc.



[SHerrington \(CC BY-SA\)](#)

Dentro de la Programación **Orientada a Objetos** ya has visto que es muy importante el concepto de **ocultación**, la cual ha sido lograda gracias a la **encapsulación** de la información dentro de las clases. De esta manera una clase puede ocultar parte de su contenido o restringir el acceso a él para evitar que sea manipulado de manera inadecuada. Los **modificadores de acceso** en Java permiten especificar el **ámbito de visibilidad** de los miembros de una clase, proporcionando así un mecanismo de accesibilidad a varios niveles.

Acabas de estudiar que cuando se definen los miembros de una clase (atributos o métodos), e incluso la propia clase, se indica (aunque sea por omisión) un modificador de acceso. En función de la visibilidad que se deseé que tengan los objetos o los miembros de esos objetos se elegirá alguno de los modificadores de acceso que has estudiado. Ahora que ya sabes cómo escribir una clase completa (declaración de la clase, declaración de sus atributos y declaración de sus métodos), vamos a hacer un repaso general de las opciones de **visibilidad (control de acceso)** que has estudiado.

Los modificadores de acceso determinan si una clase puede utilizar determinados miembros (acceder a atributos o invocar miembros) de otra clase. Existen dos niveles de control de acceso:

1. **A nivel general (nivel de clase):** visibilidad de la propia clase.
2. **A nivel de miembros:** especificación, miembro por miembro, de su nivel de visibilidad.

En el caso de la clase, ya estudiaste que los niveles de visibilidad podían ser:

- ✓ **Público** (modificador `public`), en cuyo caso la clase era visible a cualquier otra clase (cualquier otro fragmento de código del programa).
- ✓ **Privada al paquete** (sin modificador o modificador "por omisión"). En este caso, la clase sólo será visible a las demás clases del mismo paquete, pero no al resto del código del programa (otros paquetes).

En el caso de los miembros, disponías de otras dos posibilidades más de niveles de accesibilidad, teniendo un total de cuatro opciones a la hora de definir el control de acceso al miembro:

- ✓ **Público** (modificador `public`), igual que en el caso global de la clase y con el mismo significado (miembro visible desde cualquier parte del código).
- ✓ **Privado al paquete** (sin modificador), también con el mismo significado que en el caso de la clase (miembro visible sólo desde clases del mismo paquete, ni siquiera será visible desde una subclase salvo si ésta está en el mismo paquete).
- ✓ **Privado** (modificador `private`), donde sólo la propia clase tiene acceso al miembro.
- ✓ **Protegido** (modificador `protected`)

## Para saber más

Puedes echar un vistazo al artículo sobre el control de acceso a los miembros de una clase Java en los manuales de Oracle (en inglés):

[Controlling Access to Members of a Class.](#)

## Autoevaluación

**Si queremos que un atributo de una clase sea accesible solamente desde el código de la propia clase o de aquellas clases que hereden de ella, ¿qué modificador de acceso deberíamos utilizar?**

- `private`.
- `protected`.
- `public`.
- Ninguno de los anteriores.

¡Incorrecto! El modificador `private` hará que sólo el código de la propia clase tenga acceso al miembro, pero no el de clases heredadas (subclases).

¡Correcto! En efecto, el modificador `protected` es el que permite que una miembro sea accesible tanto desde la clase como desde clases heredadas.

¡Has fallado! El modificador `public` haría que ese miembro fuera accesible desde cualquier parte del programa.

¡Error! Alguno de los anteriores sí es correcto.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 5.1.- Ocultación de atributos. Métodos de acceso.



KoppCorentin (CC BY-NC-ND)

Los atributos de una clase suelen ser declarados como privados a la clase o, como mucho, `protected` (accesibles también por clases heredadas), pero no como `public`. De esta manera puedes evitar que sean manipulados inadecuadamente (por ejemplo modificarlos sin ningún tipo de control) desde el exterior del objeto.

En estos casos lo que se suele hacer es declarar esos atributos como privados o protegidos y crear métodos públicos que permitan acceder a esos atributos. Si se trata de un atributo cuyo contenido puede ser observado pero no modificado directamente, puede implementarse un método de "obtención" del atributo (en inglés se les suele llamar método de tipo `get`) y si el atributo puede ser modificado, puedes también implementar otro método para la modificación o "establecimiento" del valor del atributo (en inglés se le suele llamar método de tipo `set`). Esto ya lo has visto en apartados anteriores.

Si recuerdas la clase `Punto` que hemos utilizado como ejemplo, ya hiciste algo así con los métodos de obtención y establecimiento de las coordenadas:

```
private int x, y;  
  
// Métodos get  
  
public int obtenerX () { return x; }  
  
public int obtenerY () { return y; }  
  
// Métodos set  
  
public void establecerX (int x) { this.x= x; }  
  
public void establecerY (int y) { this.y= y; }
```

Así, para poder obtener el valor del atributo `x` de un objeto de tipo `Punto` será necesario utilizar el método `obtenerX()` y no se podrá acceder directamente al atributo `x` del objeto.

En algunos casos los programadores directamente utilizan nombres en inglés para nombrar a estos métodos: `getX`, `getY`, `setX`, `setY`, `getNombre`, `setNombre`, `getColor`, etc.

También pueden darse casos en los que no interesa que pueda observarse directamente el valor de un atributo, sino un determinado procesamiento o cálculo que se haga con el atributo (pero no el valor original). Por ejemplo podrías tener un atributo **DNI** que almacene los 8 dígitos del DNI pero no la letra del **NIF** (pues se puede calcular a partir de los dígitos). El método de acceso para el DNI (método `getDNI`) podría proporcionar el DNI completo (es decir, el NIF, incluyendo la letra), mientras que la letra no es almacenada realmente en el atributo del objeto. Algo similar podría suceder con el **dígito de control de una cuenta bancaria**, que puede no ser almacenado en el objeto, pero sí calculado y devuelto cuando se nos pide el número de cuenta completo.

En otros casos puede interesar disponer de métodos de modificación de un atributo pero a través de un determinado procesamiento previo para por ejemplo poder controlar errores o valores inadecuados. Volviendo al ejemplo del NIF, un método para modificar un DNI (método `setDNI`) podría incluir la letra (NIF completo), de manera que así podría comprobarse si el número de DNI y la letra coinciden (es un NIF válido). En tal caso se almacenará el DNI y en caso contrario se producirá un error de validación (por ejemplo lanzando una excepción). En cualquier caso, el DNI que se almacenara sería solamente el número y no la letra (pues la letra es calculable a partir del número de DNI).

### Autoevaluación

Los atributos de una clase suelen ser declarados como `public` para facilitar el acceso y la visibilidad de los miembros de la clase. ¿Verdadero o falso?

Verdadero  Falso

**Falso**

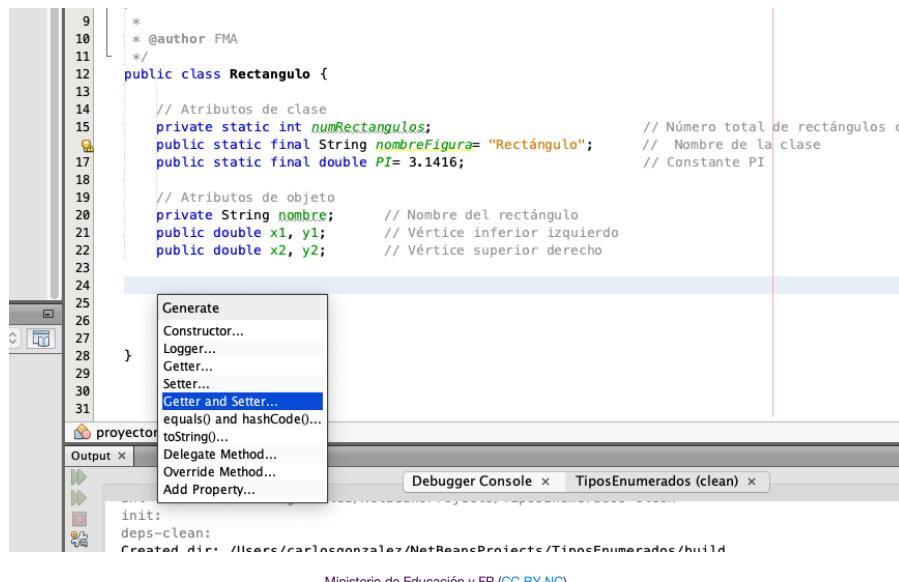
Aunque un atributo puede ser perfectamente declarado como `public`, no suele ser lo habitual. Normalmente

se oculta al resto del programa y se restringe su acceso a través de métodos de obtención y de modificación de ese atributo.

## Debes conocer

Seguimos con las utilidades de Netbeans. Has podido comprobar en el contenido que la mayoría de clases Java disponen de métodos **get** y **set** para acceder a sus atributos, que suelen ser privados. Desde el punto de visto del programador, introducir estos métodos es una tarea tediosa que nos hace ser menos productivos. Como no podía ser de otra manera, Netbeans nos ofrece la posibilidad de hacer el trabajo por nosotros.

Si para una clase que tiene una serie de atributos declarados queremos generar los métodos **get** y **set**, tan solo tenemos que activar la generación de código con la combinación de teclas **Ctrl + I** o bien desde el menú contextual seleccionando **Insert Code**. Obsérvale en la imagen:



Se seleccionas **Getter y Setter**, Netbeans te preguntará en el siguiente paso que selecciones los atributos para los cuales quieras generar los métodos.

Observa además que también podemos generar métodos constructores automáticamente, entre otros.

## 5.2.- Ocultación de métodos.

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como públicos. Pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto.

En el ejemplo anterior de objetos que contienen un DNI, será necesario calcular la letra correspondiente a un determinado número de DNI o comprobar si una determinada combinación de número y letra forman un DNI válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados de la clase (o al menos como métodos protegidos).

### Autoevaluación

Dado que los métodos de una clase forman la interfaz de comunicación de esa clase con otras clases, todos los elementos de una clase deben ser siempre declarados como públicos. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

Aunque normalmente un método suele ser una interfaz de comunicación, y en tal caso es razonable (e incluso necesario) que sea público, no todos los métodos tienen porqué ser parte de la interfaz. Es posible que en ocasiones se necesiten métodos para realizar cálculos intermedios, modificaciones temporales, adaptaciones de formatos, validaciones, comprobación de errores, etc. Este tipo de métodos tendrán sentido dentro de la clase, pero no lo tendrán fuera (o incluso puede que no sea apropiado o seguro que sean visibles desde fuera), de manera que no deberían ser declarados como `public`. Estos métodos podrían declararse como `private` o bien como `protected` (si nos interesa que sus clases descendientes también puedan acceder a esos métodos).

### Ejercicio resuelto

Vamos a intentar implementar una clase que incluya todo lo que has visto hasta ahora. Se desea crear una clase que represente un **DNI español** y que tenga las siguientes características:

- ✓ La clase almacenará el número de DNI en un `int`, sin guardar la letra, pues se puede calcular a partir del número. Este atributo será privado a la clase. Formato del atributo: `private int numDNI`.
- ✓ Para acceder al DNI se dispondrá de dos métodos `obtener` (`get`), uno que proporcionará el número de DNI (sólo las cifras numéricas) y otro que devolverá el NIF completo (incluida la letra). El formato del método será:

```
public int obtenerDNI () .
```

```
public String obtenerNIF () .
```

- ✓ Para modificar el DNI se dispondrá de dos métodos establecer (`set`), que permitirán modificar el DNI. Uno en el que habrá que proporcionar el NIF completo (número y letra). Y otro en el que únicamente será necesario proporcionar el DNI (las siete u ocho cifras). Si el DNI/NIF es incorrecto se debería lanzar algún tipo de excepción. El formato de los métodos (**sobreclaramos**) será:

```
public void establecer (String nif) throws ...
```

```
public void establecer (int dni) throws ...
```

- ✓ La clase dispondrá de algunos métodos internos privados para calcular la letra de un número de DNI cualquiera, para comprobar si un DNI con su letra es válido, para extraer la letra de un NIF, etc. Aquellos métodos que no utilicen ninguna variable de objeto podrían declararse como estáticos (pertenecientes a la clase). Formato de los métodos:

```
private static char calcularLetraNIF (int dni).  
private boolean validarNIF (String nif).  
private static char extraerLetraNIF (String nif).  
private static int extraerNumeroNIF (String nif).
```

Para calcular la letra NIF correspondiente a un número de DNI puedes consultar el artículo sobre el NIF de la Wikipedia:

[Artículo en la Wikipedia sobre el Número de Identificación Fiscal \(NIF\).](#)

[Mostrar retroalimentación](#)

En el Anexo I de esta Unidad puedes consultar toda el proceso de construcción de la clase DNI.

Puedes descargar el código fuente de la clase DNI en Java del siguiente enlace:

[Clase DNI. \(2.00 KB\)](#)

## 6.- Utilización de los métodos y atributos de una clase.

### Caso práctico



**Juan** ya ha terminado de escribir algunas de las clases de prueba para la aplicación que está comenzando a desarrollar junto con **María**. Es el momento de crear instancias de esas clases (es decir, objetos) para probar si están correctamente implementadas. La idea de **Juan** es pasar las clases a **María** junto con cierta documentación sobre su interfaz para que ella no tenga que examinar los detalles de implementación de las clases. De esta manera ella escribirá código en el que creará objetos a partir de las clases de **Juan** y a continuación comenzará a utilizar sus miembros públicos. Si todo ha ido bien, **María** habrá hecho uso de las clases de **Juan** sin tener que haber participado directamente en su desarrollo. Si se producen problemas de ejecución (de compilación no deberían producirse porque ya los habría resuelto **Juan**), **María** podrá informar de cuáles han sido esos errores para que **Juan** pueda intentar corregirlos, ya que él es quien sabrá en qué parte del código habrá que tocar.

Una vez que tienes implementada una clase con todos sus atributos y métodos, ha llegado el momento de utilizarla, es decir, de instanciar objetos de esa clase e interaccionar con ellos. En unidades anteriores ya has visto cómo declarar un objeto de una clase determinada, instanciarlo con el operador `new` y utilizar sus métodos y atributos.



[redteam \(CC BY-ND\)](#)

### Para saber más

Puedes echar un vistazo a los artículos sobre la creación y uso de objetos en Java en los manuales de Oracle (en inglés):

[Creating Objects.](#)

[Using Objects.](#)

## 6.1.- Declaración de un objeto.

Como ya has visto en unidades anteriores, la declaración de un objeto se realiza exactamente igual que la declaración de una variable de cualquier tipo:

```
<tipo> nombreVariable;
```

En este caso el tipo será alguna clase que ya hayas implementado o bien alguna de las proporcionadas por la biblioteca de Java o por alguna otra biblioteca escrita por terceros.

Por ejemplo:

```
Punto p1;  
Rectangulo r1, r2;  
Coche cocheAntonio;  
String palabra;
```

Esas variables (p1, r1, r2, cocheAntonio, palabra) en realidad son referencias (también conocidas como punteros o direcciones de memoria) que apuntan (hacen "referencia") a un objeto (una zona de memoria) de la clase indicada en la declaración.

Como ya estudiaste en la unidad dedicada a los objetos, un objeto recién declarado (referencia recién creada) no apunta a nada. Se dice que la referencia está vacía o que es una referencia nula (la variable objeto contiene el valor `null`). Es decir, la variable existe y está preparada para guardar una dirección de memoria que será la zona donde se encuentre el objeto al que hará referencia, pero el objeto aún no existe (no ha sido creado o instanciado). Por tanto se dice que apunta a un objeto nulo o inexistente.

Para que esa variable (referencia) apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que se ha reservado espacio para un objeto) es necesario crear o instanciar el objeto. Para ello se utiliza el operador `new`.



[Ministerio de Educación y FP \(CC BY-SA\)](#)

### Autoevaluación

**Aunque la declaración de un objeto es imprescindible para poder utilizarlo, ese objeto no existirá hasta que no se construya una instancia de la clase del objeto. Es decir, mientras la clase no sea instanciada, el objeto aún no existirá y lo único que se tendrá será una variable que contendrá un objeto vacío o nulo. ¿Verdadero o falso?**

Verdadero  Falso

#### Verdadero

Así es. Un objeto no existe (no ha sido registrado ni se ha asignado memoria para él) mientras no se produzca su creación o instanciación a partir de una clase.

## Ejercicio resuelto

Utilizando la clase **Rectangulo** implementada en ejercicios anteriores, indica como declararías tres objetos (variables) de esa clase llamados **r1**, **r2**, **r3**.

[Mostrar retroalimentación](#)

Se trata simplemente de realizar una declaración de esas tres variables:

```
Rectangulo r1;
```

```
Rectangulo r2;
```

```
Rectangulo r3;
```

También podrías haber declarado los tres objetos en la misma sentencia de declaración:

```
Rectangulo r1, r2, r3;
```

## 6.2.- Creación de un objeto.

Para poder crear un objeto (instancia de una clase) es necesario utilizar el operador `new`, el cual tiene la siguiente sintaxis:

```
nombreObjeto= new <ConstructorClase> ([listaParametros]);
```



El constructor de una clase (**ConstructorClase**) es un método especial que tiene toda clase y cuyo nombre coincide con el de la clase. Es quien se encarga de crear o construir el objeto, solicitando la reserva de memoria necesaria para los atributos e inicializándolos a algún valor si fuera necesario. Dado que el constructor es un método más de la clase, podrá tener también su lista de parámetros como tienen todos los métodos.

De la tarea de reservar memoria para la estructura del objeto (sus atributos más alguna otra información de carácter interno para el entorno de ejecución) se encarga el propio entorno de ejecución de Java. Es decir, que por el hecho de ejecutar un método constructor, el entorno sabrá que tiene que realizar una serie de tareas (solicitud de una zona de memoria disponible, reserva de memoria para los atributos, enlace de la variable objeto a esa zona, etc.) y se pondrá rápidamente a desempeñarlas.

Cuando escribas el código de una clase no es necesario que implementes el método constructor si no quieres hacerlo. Java se encarga de dotar de un constructor por omisión (también conocido como **constructor por defecto**) a toda clase. Ese constructor por omisión se ocupará exclusivamente de las tareas de reserva de memoria. Si deseas que el constructor realice otras tareas adicionales, tendrás que escribirlo tú. El constructor por omisión no tiene parámetros.

El constructor por defecto no se ve en el código de una clase. Lo incluirá el compilador de Java al compilar la clase si descubre que no se ha creado ningún método constructor para esa clase.

Algunos ejemplos de instanciación o creación de objetos podrían ser:

```
p1= new Punto ();  
r1= new Rectangulo ();  
r2= new Rectangulo;  
cocheAntonio= new Coche();  
palabra= String;
```

En el caso de los constructores, si éstos no tienen parámetros, pueden omitirse los paréntesis vacíos.

Un objeto puede ser declarado e instanciado en la misma línea. Por ejemplo:

```
Punto p1= new Punto ();
```

### Autoevaluación

**Si una clase no tiene constructor porque el programador no lo ha implementado, Java se encargará de dotar a esa clase de un constructor por defecto de manera que cualquier clase instanciable siempre tendrá al menos un constructor. ¿Verdadero o falso?**

Verdadero  Falso

#### Verdadero

Toda clase que puede ser instanciada siempre tendrá al menos el constructor por defecto proporcionado

por el compilador de Java. Esto no quiere decir que siempre se tenga el constructor por defecto, pues si ya has creado otros constructores personalizados, el compilador no lo incluirá. Tendrás que hacerlo tú explícitamente.

## Ejercicio resuelto

Ampliar el ejercicio anterior instanciando los objetos **r1**, **r2**, **r3** mediante el constructor por defecto.

[Mostrar retroalimentación](#)

Habrá que añadir simplemente una sentencia de creación o instancia (llamada al constructor mediante el operador `new`) por cada objeto que se desee crear:

```
Rectangulo r1, r2, r3;  
r1= new Rectangulo ();  
r2= new Rectangulo ();  
r3= new Rectangulo ();
```

## 6.3.- Manipulación de un objeto: utilización de métodos y atributos.

Una vez que un objeto ha sido declarado y creado (clase instanciada) ya sí se puede decir que el objeto existe en el entorno de ejecución, y por tanto que puede ser manipulado como un objeto más en el programa, haciéndose uso de sus atributos y sus métodos.

Para acceder a un miembro de un objeto se utiliza el operador **punto ()** del siguiente modo:

```
<nombreObjeto>.<nombreMiembro>
```

Donde **<nombreMiembro>** será el nombre de algún miembro del objeto (atributo o método) al cual se tenga acceso.

Por ejemplo, en el caso de los objetos de tipo **Punto** que has declarado e instanciado en los apartados anteriores, podrías acceder a sus miembros de la siguiente manera:

```
Punto p1, p2, p3;  
p1= new Punto();  
p1.x= 5;  
p1.y= 6;  
  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.x, p1.y);  
  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());  
  
p1.establecerX(25);  
  
p1.establecerX(30);  
  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
```

Es decir, colocando el operador **punto ()** a continuación del nombre del objeto y seguido del nombre del miembro al que se desea acceder.



Ministerio de Educación. Banco de Imágenes.  
Felix Vallés Calvo (CC BY-SA)

### Ejercicio resuelto

Utilizar el ejemplo de los rectángulos para crear un rectángulo **r1**, asignarle los valores  $x1=0$ ,  $y1=0$ ,  $x2=10$ ,  $y2=10$ , calcular su área y su perímetro y mostrarlos en pantalla.

[Mostrar retroalimentación](#)

Se trata de declarar e instanciar el objeto **r1**, rellenar sus atributos de ubicación (coordenadas de las esquinas), e invocar a los métodos `calcularSuperficie` y `calcularPerimetro` utilizando el operador **punto ()**. Por ejemplo:

```
Rectangulo r1= new Rectangulo();  
r1.x= 0;  
r1.y= 0;  
r2.x= 10;  
r2.y= 10;  
area= r1.calcularSuperficie();  
perimetro= r1.calcularPerimetro();
```

Por último faltaría mostrar en pantalla la información calculada.

Puedes descargar un ejemplo completo donde se instancia un objeto Rectangulo y se manipulan sus miembros desde el siguiente enlace:

[Proyecto EjemploRectangulos01.](#) (0.02 MB)

## 7.- Constructores.

### Caso práctico



ethorson (CC BY-SA)

**María y Juan** ya han creado y utilizado objetos y cuentan con algunos pequeños programas de ejemplo compuestos por varias clases además de la clase principal (la que contiene el método `main`). **Ada** ha estado revisando su trabajo y ha quedado muy satisfecha, aunque al observar la estructura de las clases les ha comentado algo que los ha dejado un poco despistados:

-Estas clases tienen muy buena pinta, aunque faltaría añadirles algunos constructores para poder mejorar su flexibilidad a la hora de instanciar objetos, ¿no creéis?

Ambos han asentido porque eran conscientes de que hasta el momento no habían estado incluyendo constructores en sus clases, estaban aprovechando el constructor por defecto que añadía el compilador.

-Parece que ha llegado el momento de añadir nuestros propios constructores -le dice **María a Juan**.

Como ya has estudiado en unidades anteriores, en el ciclo de vida de un objeto se pueden distinguir las fases de:

- ✓ Construcción del objeto.
- ✓ Manipulación y utilización del objeto accediendo a sus miembros.
- ✓ Destrucción del objeto.

Como has visto en el apartado anterior, durante la fase de construcción o instanciación de un objeto es cuando se reserva espacio en memoria para sus atributos y se inicializan algunos de ellos. Un **constructor** es un método especial con el **mismo nombre de la clase** y que se encarga de realizar este proceso.

El proceso de declaración y creación de un objeto mediante el operador `new` ya ha sido estudiado en apartados anteriores. Sin embargo las clases que hasta ahora has creado no tenían constructor. Has estado utilizando los constructores por defecto que proporciona Java al compilar la clase. Ha llegado el momento de que empieces a implementar tus propios constructores.

Los métodos constructores se encargan de llevar a cabo el proceso de creación o construcción de un objeto.

### Autoevaluación

¿Con qué nombre es conocido el método especial de una clase que se encarga de reservar espacio e inicializar atributos cuando se crea un objeto nuevo? ¿Qué nombre tendrá ese método en la clase?

- Método constructor. Su nombre dentro de la clase será `constructor`.
- Método inicializador. Su nombre dentro de la clase será el mismo nombre que tenga la clase.
- Método constructor. Su nombre dentro de la clase será el mismo nombre que tenga la clase.
- Método constructor. Su nombre dentro de la clase será `new`.

¡Error! En efecto el método es conocido como método constructor, pero al declararlo en la clase no se llama así.

¡Has fallado! El método no es conocido con el nombre de inicializador, aunque sí es cierto que su nombre será el mismo que el nombre de la clase.

¡Así es! ¡Enhorabuena!

¡No! ¡Error! En efecto el método es conocido como método constructor, pero al declararlo en la clase no

se llama new.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

## 7.1.- Concepto de constructor.



[blucarbnpinwheel \(CC BY-NC-SA\)](#)

de la clase.

Por tanto para crear un nuevo objeto es necesario realizar una llamada a un método constructor de la clase a la que pertenece ese objeto. Ese proceso se realiza mediante la utilización del operador `new`.

Hasta el momento ya has utilizado en numerosas ocasiones el operador `new` para instanciar o crear objetos. En realidad lo que estabas haciendo era una llamada al constructor de la clase para que reservara memoria para ese objeto y por tanto "crear" físicamente el objeto en la memoria (dotarlo de existencia física dentro de la memoria del ordenador). Dado que en esta unidad estás ya definiendo tus propias clases, parece que ha llegado el momento de que empieces a escribir también los constructores de tus clases.

Por otro lado, si un constructor es al fin y al cabo una especie de método (aunque algo especial) y Java soporta la sobrecarga de métodos, podrías plantearte la siguiente pregunta: ¿podrá una clase disponer de más de constructor? En otras palabras, ¿será posible la sobrecarga de constructores? La respuesta es afirmativa.

Una misma clase puede disponer de varios constructores. Los constructores soportan la sobrecarga.

Es necesario que toda clase tenga al menos un constructor. Si no se define ningún constructor en una clase, el compilador creará por nosotros un constructor por defecto vacío que se encarga de inicializar todos los atributos a sus valores por defecto (0 para los numéricos, `null` para las referencias, `false` para los `boolean`, etc.).

Algunas analogías que podrías imaginar para representar el constructor de una clase podrían ser:

- ✓ Los moldes de cocina para flanes, galletas, pastas, etc.
- ✓ Un cubo de playa para crear castillos de arena.
- ✓ Un molde de un lingote de oro.
- ✓ Una bolsa para hacer cubitos de hielo.



[eric.delcroix \(CC BY-NC-SA\)](#)

Una vez que incluyas un constructor personalizado a una clase, el compilador ya no incluirá el constructor por defecto (sin parámetros) y por tanto si intentas usarlo se produciría un error de compilación. Si quieras que tu clase tenga también un constructor sin parámetros tendrás que escribir su código (ya no lo hará por ti el compilador).

## 7.2.- Creación de constructores.

Cuando se escribe el código de una clase normalmente se pretende que los objetos de esa clase se creen de una determinada manera. Para ello se definen uno o más constructores en la clase. En la definición de un constructor se indican:

- ✓ El tipo de acceso.
- ✓ El nombre de la clase (el nombre de un método constructor es siempre el nombre de la propia clase).
- ✓ La lista de parámetros que puede aceptar.
- ✓ Si lanza o no excepciones.
- ✓ El cuerpo del constructor (un bloque de código como el de cualquier método).

Como puedes observar, la estructura de los constructores es similar a la de cualquier método, con las excepciones de que **no tiene tipo de dato devuelto** (no devuelve ningún valor) y que **el nombre del método constructor debe ser obligatoriamente el nombre de la clase**.

### Reflexiona

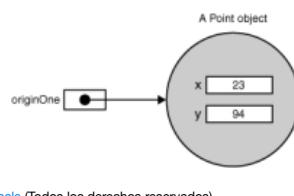
Si defines constructores personalizados para una clase, el constructor por defecto (sin parámetros) para esa clase deja de ser generado por el compilador, de manera que tendrás que crearlo tú si quieres poder utilizarlo.

Si se ha creado un constructor con parámetros y no se ha implementado el constructor por defecto, el intento de utilización del constructor por defecto producirá un error de compilación (el compilador no lo hará por nosotros).

Un ejemplo de constructor para la clase `Punto` podría ser:

```
public Punto (int x, int y)
{
    this.x= x;
    this.y= y;
    cantidadPuntos++; // Suponiendo que tengamos un atributo estático cantidadPuntos
}
```

En este caso el constructor recibe dos parámetros. Además de reservar espacio para los atributos (de lo cual se encarga automáticamente Java), también asigna sendos valores iniciales a los atributos `x` e `y`. Por último incrementa un atributo (probablemente estático) llamado `cantidadPuntos`.



Oracle (Todos los derechos reservados)

### Autoevaluación

**El constructor por defecto (sin parámetros) está siempre disponible para usarlo en cualquier clase. ¿Verdadero o falso?**

Verdadero  Falso

**Falso**

Si al implementar la clase no se ha incluido ningún constructor, el compilador añadirá un constructor por defecto. Pero si se ha proporcionado al menos un constructor, entonces el compilador no añadirá nada y no habrá constructor por defecto a no ser que lo creemos nosotros explícitamente.

## 7.3.- Utilización de constructores.

Una vez que dispongas de tus propios constructores personalizados, la forma de utilizarlos es igual que con el constructor por defecto (mediante la utilización de la palabra reservada `new`) pero teniendo en cuenta que si has declarado parámetros en tu método constructor, tendrás que llamar al constructor con algún valor para esos parámetros.

Un ejemplo de utilización del constructor que has creado para la clase Punto en el apartado anterior podría ser:

```
Punto p1;  
p1= new Punto (10, 7);
```

En este caso no se estaría utilizando el constructor por defecto sino el constructor que acabas de implementar en el cual además de reservar memoria se asigna un valor a algunos de los atributos.



[idITE \(CC BY-NC-SA\)](#)

### Para saber más

Puedes echar un vistazo al artículo sobre constructores de una clase Java en los manuales de Oracle (en inglés):

[Providing Constructors for Your Classes.](#)

También puedes echar un vistazo a este vídeo que ya se te han recomendado en unidades anteriores. Ahora probablemente comprenderás mucho mejor el proceso que se muestra pues es más o menos lo que has tenido que hacer tú:

<https://www.youtube.com/embed/k92WaQyzVd4>

[Resumen textual alternativo](#)

### Ejercicio resuelto

Ampliar el ejercicio de la clase **Rectangulo** añadiéndole tres constructores:

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1);
2. Un constructor con cuatro parámetros, **x1, y1, x2, y2**, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
3. Un constructor con dos parámetros, **base** y **altura**, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.

[Mostrar retroalimentación](#)

En el caso del primer constructor lo único que hay que hacer es "rellenar" los atributos x1, y1, x2, y2 con los valores 0, 0, 1, 1:

```
public Rectangulo ()  
{  
    x1= 0.0;  
  
    y1= 0.0;  
  
    x2= 1.0;  
  
    y2= 1.0;  
}
```

Para el segundo constructor es suficiente con asignar a los atributos x1, y1, x2, y2 los valores de los parámetros x1, y1, x2, y2. Tan solo hay que tener en cuenta que al tener los mismos nombres los parámetros del método que los atributos de la clase, estos últimos son ocultados por los primeros y para poder tener acceso a ellos tendrás que utilizar el operador de autorreferencia `this`:

```
public Rectangulo (double x1, double y1, double x2, double y2)  
{  
  
    this.x1= x1;  
  
    this.y1= y1;  
  
    this.x2= x2;  
  
    this.y2= y2;  
}
```

En el caso del tercer constructor tendrás que inicializar el vértice (x1, y1) a (0,0) y el vértice (x2,y2) a (0 + base, 0 + altura), es decir a (base, altura):

```
public Rectangulo (double base, double altura)  
{  
  
    this.x1= 0.0;  
  
    this.y1= 0.0;  
  
    this.x2= base;  
  
    this.y2= altura;  
}
```

Puedes descargar un ejemplo completo donde se amplía la clase `Rectangulo` con esos tres constructores y se utilizan en un programa de prueba desde el siguiente enlace:

[Proyecto EjemploRectangulos02.](#)

## 7.4.- Constructores de copia.

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas). Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto.

Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales (basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si pudieras **clonar** el objeto tantas veces como te haga falta. A este tipo de mecanismo se le suele llamar **constructor copia** o **constructor de copia**.

Un constructor copia es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

Un ejemplo de constructor copia para la clase **Punto** podría ser:

```
public Punto (Punto p)
{
    this.x= p.obtenerX();
    this.y= p.obtenerY();
}
```

En este caso el constructor recibe como parámetro un objeto del mismo tipo que el que va a ser creado (clase **Punto**), inspecciona el valor de sus atributos (atributos **x** e **y**), y los reproduce en los atributos del objeto en proceso de construcción (**this**).

Un ejemplo de utilización de ese constructor podría ser:

```
Punto p1, p2;
p1= new Punto (10, 7);
p2= new Punto (p1);
```

En este caso el objeto **p2** se crea a partir de los valores del objeto **p1**.

### Autoevaluación

Toda clase debe incluir un constructor copia en su implementación. ¿Verdadero o falso?

- Verdadero  Falso

#### Falso

No tiene porqué. Puede resultar muy útil disponer de uno o varios constructores copia pero no es algo imperativo. Para determinado tipo de objetos podrá ser casi imprescindible disponer de constructores copia para poder utilizarlos eficientemente, pero en otro tipo de objetos es posible que no sea necesario. Dependerá del caso concreto.

## Ejercicio resuelto

Ampliar el ejercicio de la clase `Rectangulo` añadiéndole un constructor copia.

[Mostrar retroalimentación](#)

Se trata de añadir un nuevo constructor además de los tres que ya habíamos creado:

```
// Constructor copia  
  
public Rectangulo (Rectangulo r) {  
  
    this.x1= r.x1;  
  
    this.y1= r.y1;  
  
    this.x2= r.x2;  
  
    this.y2= r.y2;  
  
}
```

Para usar este constructor basta con haber creado anteriormente otro `Rectangulo` para utilizarlo como base de la copia. Por ejemplo:

```
Rectangulo r1, r2;  
  
r1= new Rectangulo (0,0,2,2);  
  
r2= new Rectangulo (r1);
```

## 7.5.- Destrucción de objetos.

Como ya has estudiado en unidades anteriores, cuando un objeto deja de ser utilizado, los recursos usados por él (memoria, acceso a archivos, conexiones con bases de datos, etc.) deberían de ser liberados para que puedan volver a ser utilizados por otros procesos (mecanismo de **destrucción del objeto**).

Mientras que de la construcción de los objetos se encargan los métodos constructores, de la destrucción se encarga un proceso del entorno de ejecución conocido como **recolector de basura (garbage collector)**. Este proceso va buscando periódicamente objetos que ya no son referenciados (no hay ninguna variable que haga referencia a ellos) y los marca para ser eliminados. Posteriormente los irá eliminando de la memoria cuando lo considere oportuno (en función de la carga del sistema, los recursos disponibles, etc.).

Normalmente se suele decir que en Java no hay método destructor y que en otros lenguajes orientados a objetos como **C++**, sí se implementa explícitamente el destructor de una clase de la misma manera que se define el constructor. En realidad en Java también es posible implementar el método destructor de una clase, se trata del método `finalize()`.

Este método `finalize` es llamado por el recolector de basura cuando va a destruir el objeto (lo cual nunca se sabe cuándo va a suceder exactamente, pues una cosa es que el objeto sea marcado para ser borrado y otra que sea borrado efectivamente). Si ese método no existe, se ejecutará un destructor por defecto (el método `finalize` que contiene la clase `Object`, de la cual heredan todas las clases en Java) que liberará la memoria ocupada por el objeto. Se recomienda por tanto que si un objeto utiliza determinados recursos de los cuales no tienes garantía que el entorno de ejecución los vaya a liberar (cerrar archivos, cerrar conexiones de red, cerrar conexiones con bases de datos, etc.), implementes explícitamente un método `finalize` en tus clases. Si el único recurso que utiliza tu clase es la memoria necesaria para albergar sus atributos, eso sí será liberado sin problemas. Pero si se trata de algo más complejo, será mejor que te encargues tú mismo de hacerlo implementando tu destructor personalizado (`finalize`).

Por otro lado, esta forma de funcionar del entorno de ejecución de Java (destrucción de objetos no referenciados mediante el recolector de basura) implica que no puedes saber exactamente cuándo un objeto va a ser definitivamente destruido, pues si una variable deja de ser referenciada (se cierra el ámbito de ejecución donde fue creada) no implica necesariamente que sea inmediatamente borrada, sino que simplemente es marcada para que el recolector la borre cuando pueda hacerlo.

Si en un momento dado fuera necesario garantizar que el proceso de finalización (método `finalize`) sea invocado, puedes recurrir al método `runFinalization()` de la clase `System` para forzarlo:

```
System.runFinalization();
```

Este método se encarga de llamar a todos los métodos `finalize` de todos los objetos marcados por el recolector de basura para ser destruidos.

Si necesitas implementar un destructor (normalmente no será necesario), debes tener en cuenta que:

- ✓ El nombre del método destructor debe ser `finalize()`.
- ✓ No puede recibir parámetros.
- ✓ Sólo puede haber un destructor en una clase. No es posible la sobrecarga dado que no tiene parámetros.
- ✓ No puede devolver ningún valor. Debe ser de tipo `void`.

### Autoevaluación

¿Cuando se abandona el ámbito de un objeto en Java éste es marcado por el recolector de basura para ser destruido. En muchas ocasiones una clase Java no tiene un método destructor, pero si fuera necesario hacerlo ¿podrías implementar un método destructor en una clase Java? ¿Qué nombre habría que ponerle?

- Sí es posible. El nombre del método sería `finalize()`.
- No es posible disponer de un método destructor en una clase Java.
- Sí es posible. El nombre del método sería `destructor()`.
- Sí es posible. El nombre del método sería `~nombreClase`, como en el lenguaje C++.

¡Así es! ¡Enhorabuena!

¡Error! Sí se puede tener un método destructor.

¡Has fallado! En efecto se puede implementar explícitamente un método destructor, aunque ése no sería el nombre.

¡Te has equivocado! Aunque es cierto que se puede implementar explícitamente un método destructor, ése no sería el nombre del destructor.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 8.- Empaquetado de clases.

### Caso práctico



[lemonhalf \(CC BY-NC-SA\)](#)

**María** y **Juan** ya han terminado por ahora de escribir todas las clases que habían diseñado. Es el momento de organizar adecuadamente todo el código que tienen implementado a lo largo de todas esas clases. **María** recuerda que hace algunos días Juan ya le habló de la posibilidad de organizar las clases en paquetes:

Este sería un buen momento para poner en práctica todo aquello que me enseñaste sobre los paquetes, ¿no?

-La verdad es que tienes razón. Esto hay que organizarlo un poco...-Le contesta

**Juan.**

La **encapsulación** de la información dentro de las clases ha permitido llevar a cabo el proceso de ocultación, que es fundamental para el trabajo con clases y objetos. Es posible que conforme vaya aumentando la complejidad de tus aplicaciones necesites que algunas de tus clases puedan tener acceso a parte de la implementación de otras debido a las relaciones que se establezcan entre ellas a la hora de diseñar tu modelo de datos. En estos casos se puede hablar de un nivel superior de encapsulamiento y ocultación conocido como **empaquetado**.

Un **paquete** consiste en un conjunto de clases relacionadas entre sí y agrupadas bajo un mismo nombre. Normalmente se encuentran en un mismo paquete todas aquellas clases que forman una biblioteca o que reúnen algún tipo de característica en común. Esto la organización de las clases para luego localizar fácilmente aquellas que vayas necesitando.

## 8.1.- Jerarquía de paquetes.

Los paquetes en Java pueden organizarse jerárquicamente de manera similar a lo que puedes encontrar en la estructura de carpetas en un dispositivo de almacenamiento, donde:

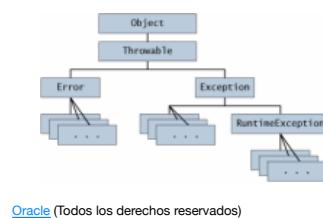
- ✓ Las clases serían como los archivos.
- ✓ Cada paquete sería como una carpeta que contiene archivos (clases).
- ✓ Cada paquete puede además contener otros paquetes (como las carpetas que contienen carpetas).
- ✓ Para poder hacer referencia a una clase dentro de una estructura de paquetes, habrá que indicar la trayectoria completa desde el paquete raíz de la jerarquía hasta el paquete en el que se encuentra la clase, indicando por último el nombre de la clase (como el path absoluto de un archivo).

La estructura de paquetes en Java permite organizar y clasificar las clases, evitando conflictos de nombres y facilitando la ubicación de una clase dentro de una estructura jerárquica.

Por otro lado, la organización en paquetes permite también el control de acceso a miembros de las clases desde otras clases que estén en el mismo paquete gracias a los modificadores de acceso (recuerda que uno de los modificadores que viste era precisamente el de paquete).

Las clases que forman parte de la jerarquía de clases de Java se encuentran organizadas en diversos paquetes.

Todas las clases proporcionadas por Java en sus bibliotecas son miembros de distintos paquetes y se encuentran organizadas jerárquicamente. Dentro de cada paquete habrá un conjunto de clases con algún tipo de relación entre ellas. Se dice que todo ese conjunto de paquetes forman la API de Java. Por ejemplo las clases básicas del lenguaje se encuentran en el paquete `java.lang`, las clases de entrada/salida las podrás encontrar en el paquete `java.io` y en el paquete `java.math` podrás observar algunas clases para trabajar con números grandes y de gran precisión.



Oracle (Todos los derechos reservados)

### Recomendación

En la unidad dedicada a la utilización de objetos (Unidad 3), se hizo una descripción general de los paquetes más importantes ofrecidos por el lenguaje Java. Puede ser un buen momento para volver a echarles un vistazo.

### Para saber más

Puedes echar un vistazo a toda la jerarquía de paquetes de la API de Java en la documentación oficial de Oracle (en inglés):

[Java Platform, Standard Edition 14 API Specification.](#)

Se trata de la documentación de referencia del lenguaje. Es documentación que se consulta, no es necesario aprenderse nada aunque poco a poco, con la práctica, iremos conociendo más clases de este API.

## 8.2.- Utilización de los paquetes.



Es posible acceder a cualquier clase de cualquier paquete (siempre que ese paquete esté disponible en nuestro sistema, obviamente) mediante la calificación completa de la clase dentro de la estructura jerárquica de paquete. Es decir indicando la trayectoria completa de paquetes desde el paquete raíz hasta la propia clase. Eso se puede hacer utilizando el operador **punto.**) para especificar cada subpaquete:

```
paquete_raiz.subpaquete1.subpaquete2. .... .subpaquete_n.NombreClas
```

[idITE \(CC BY-NC-SA\)](#)

Por ejemplo:

```
java.lang.String.
```

En este caso se está haciendo referencia a la clase **String** que se encuentra dentro del paquete **java.lang**. Este paquete contiene las clases elementales para poder desarrollar una aplicación Java.

Otro ejemplo podría ser:

```
java.util.regex.Patern.
```

En este otro caso se hace referencia a la clase **Patern** ubicada en el paquete **java.util.regex**, que contiene clases para trabajar con expresiones regulares.

Dado que puede resultar bastante tedioso tener que escribir la trayectoria completa de una clase cada vez que se quiera utilizar, existe la posibilidad de indicar que se desea trabajar con las clases de uno o varios paquetes. De esa manera cuando se vaya a utilizar una clase que pertenezca a uno de esos paquetes no será necesario indicar toda su trayectoria. Para ello se utiliza la sentencia **import** (importar):

```
import paquete_raiz.subpaquete1.subpaquete2. .... .subpaquete_n.NombreClase;
```

De esta manera a partir de ese momento podrá utilizarse directamente **NombreClase** en lugar de toda su trayectoria completa.

Los ejemplos anteriores quedarían entonces:

```
import java.lang.String;  
import java.util.regex.Patern;
```

Si suponemos que vamos a utilizar varias clases de un mismo paquete, en lugar de hacer un **import** de cada una de ellas, podemos utilizar el **comodín** (símbolo **asterisco: "..."**) para indicar que queremos importar todas las clases de ese paquete y no sólo una determinada:

```
import java.lang.*;  
import java.util.regex.*;
```

Si un paquete contiene subpaquetes, el comodín no importará las clases de los subpaquetes, tan solo las que haya en el paquete. La importación de las clases contenidas en los subpaquetes habrá que indicarla explícitamente. Por ejemplo:

```
import java.util.*;  
import java.util.regex.*;
```

En este caso se importarán todas las clases del paquete `java.util` (clases `Date`, `Calendar`, `Timer`, etc.) y de su subpaquete `java.util.regex` (`Matcher` y `Pattern`), pero las de otros subpaquetes como `java.util.concurrent` O `java.util.jar`.

Por último tan solo indicar que en el caso del paquete `java.lang`, no es necesario realizar importación. El compilador, dada la importancia de este paquete, permite el uso de sus clases sin necesidad de indicar su trayectoria (es como si todo archivo Java incluyera en su primera línea la sentencia `import java.lang`\*).

## Autoevaluación

La sentencia `import` nos facilita las cosas a la hora de especificar las clases que queremos utilizar en nuestro archivo Java. Con el uso del comodín (asterisco) podemos importar todas las clases y subpaquetes que se encuentran en un determinado paquete a través de una sola sentencia `import`. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

Es cierto que se importarán todas las clases incluidas en el paquete especificado, pero no sucederá lo mismo con los subpaquetes. Será necesaria también una sentencia `import` para cada subpaquete.

## 8.3.- Inclusión de una clase en un paquete.

---

Al principio de cada archivo .java se puede indicar a qué paquete pertenece mediante la palabra reservada `package` seguida del nombre del paquete:

```
package nombre_paquete
```

Por ejemplo:

```
package paqueteEjemplo;  
  
class claseEjemplo {  
  
    ...  
  
}
```

La sentencia `package` debe ser incluida en cada archivo fuente de cada clase que quieras incluir ese paquete. Si en un archivo fuente hay definidas más de una clase, todas esas clases formarán parte del paquete indicado en la sentencia `package`.

Si al comienzo de un archivo Java no se incluyen ninguna sentencia `package`, el compilador considerará que las clases de ese archivo formarán parte del paquete por omisión (un paquete sin nombre asociado al proyecto).

Para evitar la ambigüedad, dentro de un mismo paquete no puede haber dos clases con el mismo nombre, aunque sí pueden existir clases con el mismo nombre si están en paquetes diferentes. El compilador será capaz de distinguir una clase de otra gracias a que pertenecen a paquetes distintos.

Como ya has visto en unidades anteriores, el nombre de un archivo fuente en Java se construye utilizando el nombre de la clase pública que contiene junto con la extensión .java, pudiendo haber únicamente una clase pública por cada archivo fuente. El nombre de la clase debía coincidir (en mayúsculas y minúsculas) exactamente con el nombre del archivo en el que se encontraba definida. Así, si por ejemplo tenías una clase `Punto` dentro de un archivo `Punto.java`, la compilación daría lugar a un archivo `Punto.class`.

En el caso de los paquetes, la correspondencia es a nivel de directorios o carpetas. Es decir, si la clase `Punto` se encuentra dentro del paquete `prog.figuras`, el archivo `Punto.java` debería encontrarse en la carpeta `prog\figuras`. Para que esto funcione correctamente el compilador ha de ser capaz de localizar todos los paquetes (tanto los estándar de Java como los definidos por otros programadores). Es decir, que el compilador debe tener conocimiento de dónde comienza la estructura de carpetas definida por los paquetes y en la cual se encuentran las clases. Para ello se utiliza el `ClassPath` cuyo funcionamiento habrás estudiado en las primeras unidades de este módulo. Se trata de una variable de entorno que contiene todas las rutas en las que comienzan las estructuras de directorios (distintas jerarquías posibles de paquetes) en las que están contenidas las clases.

## 8.4.- Proceso de creación de un paquete.

---

Para crear un paquete en Java te recomendamos seguir los siguientes pasos:

**oñer un nombre al paquete.** Suele ser habitual utilizar el dominio de Internet de la empresa que ha creado el paquete. Por ejemplo, para el caso de **miempresa.com**, podría utilizarse un nombre de paquete **com.miemuestra**.

1. **Crear una estructura jerárquica de carpetas equivalente a la estructura de subpaquetes.** La ruta de la raíz de esa estructura jerárquica deberá estar especificada en el **ClassPath** de Java.
2. **Especificiar a qué paquete pertenecen la clase** (o clases) de el archivo .java mediante el uso de la sentencia **package** tal y como has visto en el apartado anterior.

Este proceso ya lo has debido de llevar a cabo en unidades anteriores al compilar y ejecutar clases con paquetes. Estos pasos simplemente son para que te sirvan como recordatorio del procedimiento que debes seguir a la hora de clasificar, jerarquizar y utilizar tus propias clases.

# Anexo I.- Proceso de implementación de la clase DNI.

## Ejercicio resuelto

Vamos a intentar implementar una clase que incluya todo lo que has visto hasta ahora. Se desea crear una clase que represente un **DNI español** y que tenga las siguientes características:

- ✓ La clase almacenará el número de DNI en un `int`, sin guardar la letra, pues se puede calcular a partir del número. Este atributo será privado a la clase. Formato del atributo: `private int numDNI`.
- ✓ Para acceder al DNI se dispondrá de dos métodos **obtener** (`get`), uno que proporcionará el número de DNI (sólo las cifras numéricas) y otro que devolverá el NIF completo (incluida la letra). El formato del método será:

```
public int obtenerDNI ().

public String obtenerNIF ().
```

- ✓ Para modificar el DNI se dispondrá de dos métodos establecer (`set`), que permitirán modificar el DNI. Uno en el que habrá que proporcionar el NIF completo (número y letra). Y otro en el que únicamente será necesario proporcionar el DNI (las siete u ocho cifras). Si el DNI/NIF es incorrecto se debería lanzar algún tipo de **excepción**. El formato de los métodos (**sobrecargados**) será:

```
public void establecer (String nif) throws ...
public void establecer (int dni) throws ...
```

- ✓ La clase dispondrá de algunos de métodos internos privados para calcular la letra de un número de DNI cualquiera, para comprobar si un DNI con su letra es válido, para extraer la letra de un NIF, etc. Aquellos métodos que no utilicen ninguna variable de objeto podrían declararse como estáticos (pertenecientes a la clase). Formato de los métodos:

```
private static char calcularLetraNIF (int dni).

private boolean validarNIF (String nif).

private static char extraerLetraNIF (String nif).

private static int extraerNumeroNIF (String nif).
```

[Mostrar retroalimentación](#)

La clase tendrá un único atributo de objeto: **el número de DNI**.

```
private int numDNI;
```

Está claro que para poder trabajar con los DNI/NIF vas a necesitar implementar el algoritmo para calcular la letra de un número de DNI. Para ello puedes crear un método (que en principio podría ser privado) que realice ese cálculo. Para facilitar la implementación de ese método, crearemos un `array` estático y constante (`final`) con las letras posibles que puede tener un NIF y en el orden adecuado para la aplicación del algoritmo de cálculo de la letra (algoritmo conocido como **módulo 23**):

```
private static final String LETRAS_DNI= "TRWAGMYFPDXBNJZSQVHLCKE";
```

Con esta cadena disponible, es muy sencillo implementar el algoritmo del **módulo 23**:

```
private static char calcularLetraNIF (int dni) {  
    char letra;  
  
    // Cálculo de la letra NIF  
  
    letra= LETRAS_DNI.charAt(dni % 23);  
  
    // Devolución de la letra NIF  
  
    return letra;  
}
```

Este método estático ha sido definido como privado, aunque también podría haber sido definido como público para que otros objetos pudieran hacer uso de él (típico ejemplo de uso de un método estático).

Para poder manipular adecuadamente la cadena NIF, podemos crear un par de métodos para extraer el número de DNI o la letra a partir de una cadena NIF. Ambos métodos pueden declararse estáticos y privados (aunque no es la única posibilidad):

```
private static char extraerLetraNIF (String nif) {  
  
    char letra= nif.charAt(nif.length()-1);  
  
    return letra;  
}  
  
private static int extraerNumeroNIF (String nif) {  
  
    int numero= Integer.parseInt(nif.substring(0, nif.length()-1));  
  
    return numero;  
}
```

Una vez que disponemos de todos estos métodos es bastante sencillo escribir un método de comprobación de la validez de un NIF:

- ✓ Extracción del número.
- ✓ Extracción de la letra.
- ✓ Cálculo de la letra a partir del número.
- ✓ Comparación de la letra extraída con la letra calculada.

De manera que el método nos podría quedar:

```
private static boolean validarNIF (String nif) {  
  
    boolean valido= true; // Suponemos el NIF válido mientras no se encuentre algún fallo  
  
    char letra_calculada;  
  
    char letra_leida;  
  
    int dni_leido;  
  
    if (nif == null) { // El parámetro debe ser un objeto no vacío  
  
        valido= false;  
    }  
  
    else if (nif.length()<8 || nif.length()>9) { // La cadena debe estar entre 8(7+1) y 9(8+1) caracteres  
  
        valido= false;  
    }
```

```

    }

    else {

        letra_leida= DNI.extraerLetraNIF (nif);      // Extraemos la letra de NIF (letra)

        dni_leido= DNI.extraerNumeroNIF (nif); // Extraemos el número de DNI (int)

        letra_calculada= DNI.calcularLetraNIF(dni_leido); // Calculamos la letra de NIF a partir de

        if (letra_leida == letra_calculada) { // Comparamos la letra extraída con la calculada

            // Todas las comprobaciones han resultado válidas. El NIF es válido.

            valido= true;

        }

        else {

            valido= false;

        }

    }

    return valido;

}

```

En el código de este método puedes comprobar que se hace uso de los métodos estáticos colocando explícitamente el nombre de la clase:

```

DNI.extraerLetraNIF.

DNI.extraerNumeroNIF.

DNI.calcularLetraNIF.

```

En realidad en este caso no habría sido necesario pues estamos en el interior de la clase, pero si finalmente hubiéramos decidido hacer públicos estos métodos, así es como habría que llamarlos desde fuera (usando el nombre de la clase y no el de una instancia).

Y por último tan solo quedarían por implementar los métodos públicos (la interfaz):

- ✓ Los dos métodos **obtener** (*get*). Obtener el NIF (*String*) u obtener el DNI (*int*).
- ✓ Los dos métodos **establecer** (*set*). A partir de un *int* y a partir de un *String*.

En el primer caso habrá que devolver información añadiéndole (si es necesario) información adicional calculada, y en el segundo habrá que realizar una serie de comprobaciones antes de proceder a almacenar el nuevo valor de DNI/NIF.

El código de los métodos **obtener** podría quedar así:

```

public String obtenerNIF () {

    // Variables locales

    String cadenaNIF; // NIF con letra para devolver

    char letraNIF; // Letra del número de NIF calculado

    // Cálculo de la letra del NIF

    letraNIF= DNI.calcularLetraNIF (numDNI);

    // Construcción de la cadena del DNI: número + letra

    cadenaNIF= Integer.toString(numDNI) + String.valueOf(letraNIF);

    // Devolución del resultado
}

```

```
        return cadenaNIF;
    }

    public int obtenerDNI () {
        return numDNI;
    }
```

En el caso de los métodos establecer (**método establecer sobrecargado**) podemos lanzar una excepción básica con un mensaje de error de "NIF/DNI inválido" para que la reciba el objeto que utilice este método. De esta manera podría controlarse el error de un posible establecimiento de valores de NIF/DNI inválido.

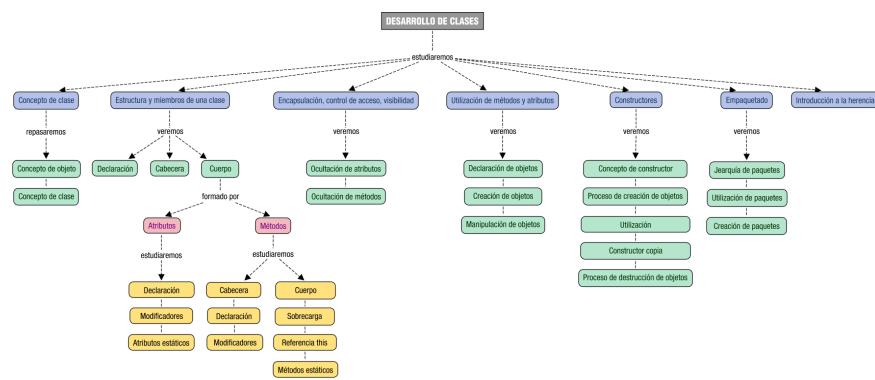
El código de los métodos **establecer** podría quedar así;

```
public void establecer (String nif) throws Exception {
    if (validarNIF (nif)) { // Valor válido: lo almacenamos
        this.numDNI= DNI.extraerNumeroNIF(nif);
    }
    else { // Valor inválido: lanzamos una excepción
        throw new Exception ("NIF inválido: " + nif);
    }
}

public void establecer (int dni) throws Exception {
    // Comprobación de rangos
    if (dni>999999 && dni<99999999) {
        this.numDNI= dni; // Valor válido: lo almacenamos
    }
    else { // Valor inválido: lanzamos una excepción
        throw new Exception ("DNI inválido: " + String.valueOf(dni));
    }
}
```

## 9.- Conclusiones

En la Unidad de Trabajo 3 ya hicimos una introducción a la programación orientada a objetos, sin embargo en esta unidad hemos realizado una profundización en los conceptos más importantes de este paradigma: clases y objetos. Hemos analizado que componentes contiene una clase (atributos y métodos) y cómo se definen y utilizan. Además hemos profundizado en conceptos como sobrecarga, encapsulación, parámetros, métodos estáticos, modificadores de acceso, etc. También hemos profundizado en el concepto de constructor, su definición y su uso. Por último, hemos introducido la posibilidad de organizar un proyecto en paquetes, algo indispensable en proyectos grandes. Aunque se trata realmente de un repaso a los conceptos trabajados en la Unidad 3, deben quedar claros de cara a afrontar las siguientes unidades.



También hemos introducido de nuevo el concepto avanzado de herencia, que será trabajado en profundidad en posteriores unidades junto a otros como polimorfismo.

En la siguiente unidad nos centraremos en las estructuras compuestas de datos. Hasta ahora hemos trabajado con tipos primitivos simples de datos a menudo necesitamos estructuras más complejas para almacenar información. Será en la unidad 6

# Estructuras de almacenamiento de información.

## Caso práctico



Ministerio de Educación y FP [\(CC BY-NC\)](#)

**Ana** ha recibido un pequeño encargo de parte de su tutora, **María**. Se trata de que realice un pequeño programita, muy sencillo pero fundamental.

-Hola **Ana** -dice **María**-, hoy tengo una tarea especial para ti.

-¿Sí? -responde **Ana**. Estoy deseando, últimamente no hay nada que se me resista, llevo dos semanas en racha.

-Bueno, quizás esto se te resista un poco más, es fácil, pero tiene cierta complicación. Un cliente para el que hicimos una aplicación, nos ha dicho si podemos ayudarle. El cliente tiene una aplicación para gestionar los pedidos que recibe de sus clientes. Normalmente, recibe por correo electrónico los pedidos en un formato concreto que todos sus clientes llevan tiempo usando. El cliente suele transcribir el pedido desde el correo electrónico a la aplicación, copiando dato a dato, pero nos ha preguntado si es posible que copie todo el pedido de golpe, y que la aplicación lo procese, detectando posibles errores en el pedido.

-¿Qué? -dice **María** con cierta perplejidad.

-Me alegra que te guste -dice María esbozando una sonrisa picara-, sé que te gustan los retos.

-Pero, ¿eso cómo se hace? ¿Cómo compruebo yo si el pedido es válido? ¿Y si el cliente ha puesto un producto que no existe?

-No mujer, se trata de comprobar si el pedido tiene el formato correcto y transformarlo de forma que se incorpore fácilmente a los otros pedidos que tenga el cliente en su base de datos. De la verificación de si hay algún producto que no existe, o de si hay alguna incoherencia en el pedido se encarga otra parte del software, que ya he realizado yo.

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?

Lo mismo ocurre en otros casos. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros **datos simples** (números y letras). A veces, los datos que tiene que manejar la aplicación son **datos compuestos**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, los datos podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.



ITE. Óscar Javier Estupiñán Estupiñán [\(CC BY-NC\)](#)

Los datos compuestos son un tipo de estructura de datos, y en realidad ya los has manejado. **Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos**, y el objeto en sí, la instancia de una clase, sería el dato compuesto. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

Aquí podrás aprender esas soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.



[Ministerio de Educación y Formación Profesional](#). (Dominio público)

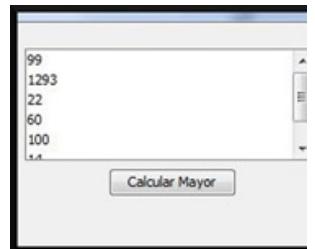
**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Introducción a las estructuras de almacenamiento.

¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo? Seguro que te resultaría fácil. Pero, ¿y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica? Entonces la cosa se complica.

Un listado de números que aumenta o decrece en tamaño es una de las cosas que aprenderás a utilizar aquí, utilizando estructuras de datos. Pasaremos por alto las clases y los objetos, pues ya los has visto con anterioridad, pero debes saber que las clases en sí mismas son la evolución de un tipo de estructuras de datos conocidas como datos compuestos (también llamadas registros). Las clases, además de aportar la ventaja de agrupar datos relacionados entre sí en una misma estructura (característica aportada por los datos compuestos), permiten agregar métodos que manejen dichos datos, ofreciendo una herramienta de programación sin igual. Pero todo esto ya lo sabías.



Salvador Romero Villegas ([CC BY-NC](#))

Las estructuras de almacenamiento, en general, se pueden clasificar de varias formas. Por ejemplo, atendiendo a si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo, se pueden distinguir:

- ✓ Estructuras con capacidad de **almacenar varios datos del mismo tipo**: varios números, varios caracteres, etc. Ejemplos de estas estructuras son los arrays, las cadenas de caracteres, las listas y los conjuntos.
- ✓ Estructuras con capacidad de **almacenar varios datos de distinto tipo**: números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las clases.

Otra forma de clasificar las estructuras de almacenamiento va en función de si pueden o no cambiar de tamaño de forma dinámica:

- ✓ **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multidimensionales).
- ✓ **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las listas, árboles, conjuntos y, como veremos también, el caso de algunos tipos de cadenas de caracteres.

Por último, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, podemos diferenciar varios tipos de estructuras:

- ✓ **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays.
- ✓ **Estructuras ordenadas**. Se trata de estructuras que al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc.

Todavía no conoces mucho de las estructuras, y probablemente todo te suena raro y extraño. No te preocupes, poco a poco irás descubriendolas. Verás que son sencillas de utilizar y muy cómodas.

## Autoevaluación

**El tamaño de las estructuras de almacenamiento siempre se determina en el momento de la creación. ¿Verdadero o falso?**

- Verdadero.
- Falso.

Incorrecto, en algunos casos sí, y en otros no. Existen estructuras dinámicas que cambian de tamaño al insertar o eliminar datos.

Efectivamente, lo has entendido a la perfección.

## Solución

1. Incorrecto
2. Opción correcta

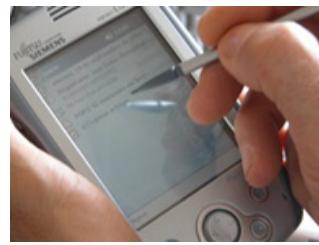
## 2.- Cadenas de caracteres.

### Caso práctico

Ana está asustada, le acaban de asignar una nueva tarea y le acaban de enviar por correo el formato del archivo que tiene que procesar. El archivo tiene un formato concreto, un tanto extraño. Primero tiene una cabecera con información general del pedido, y luego tiene los artículos del pedido, cada uno en una línea diferente. ¿Podrá realizar la tarea encomendada?. En este momento lo que le preocupa es cómo evaluar las líneas del pedido, mas adelante se encargará de estudiar cómo se puede leer el contenido del fichero y procesarlo.

La empresa recibe los pedidos siguiendo un modelo normalizado. En realidad se trata de una pequeña aplicación web que al llenar un formulario, envía un correo electrónico con la información siguiendo una estructura fija. A continuación puedes ver un ejemplo:

```
## PEDIDO ##
Número de pedido: { 20304 }
Cliente: { Muebles Bonitos S.A. }
Código del cliente: { 00293 }
Dirección de factura: { C/ De en frente, 11 }
} Dirección de entrega: { C/ De al lado, 22 }
Nombre del contacto: { Elias }
Teléfono del contacto: { 987654321 }
Correo electrónico del contacto: { mail@mail1234.com }
Fecha preferente de entrega: { 19/11/2012 }
Forma de pago: { Transferencia }
## ARTICULOS ##
{ Código Artículo | Descripción | Cantidad }
{ 0001231 | Tuercas tipo 1 | 200 }
{ 0001200 | Tornillos tipo 1 | 200 }
{ 0002200 | Arandelas tipo 2 | 200 }
## FIN ARTICULOS ##
## FIN PEDIDO ##
```



ITE idITE=160222 ([CC BY-NC](#))

Como puedes observar, el pedido tiene una estructura fija, y solamente cambian los datos que están entre llaves. El resto del texto es fijo, dado que como se ha dicho antes, el pedido lo genera una aplicación web, y lo envía por correo electrónico al cliente.

Cuando un cliente solicita un pedido a esta empresa, no puede poner en ningún campo los símbolos "{", "}", ni "|", porque entonces entraría en conflicto con el formato enviado por correo y daría problemas, así que se optó por impedir que el cliente pudiera poner dichos caracteres extraños. Esto garantiza que cada campo del pedido se va a poder leer adecuadamente.

Lo más conflictivo, es la sección de artículos del pedido, en su primera parte, tiene una especie de cabecera que ayuda a identificar que es cada una de las columnas. Y después, tiene tantas líneas como artículos tenga el pedido, si tiene 100 artículos, tendrán 100 líneas que incluirán: código del artículo, descripción del artículo y la cantidad.

Probablemente, una de las cosas que mas utilizarás cuando estés programando en cualquier lenguaje de programación son las cadenas de caracteres. Las cadenas de caracteres son estructuras de almacenamiento que permiten almacenar una secuencia de caracteres de casi cualquier longitud. Y la pregunta ahora es, ¿qué es un carácter?

ITE. Pedro García Barbudo  
([CC BY-NC](#))

En Java y en todo lenguaje de programación, y por ende, en todo sistema informático, los caracteres se codifican como secuencias de bits que representan a los símbolos usados en la comunicación humana. Estos símbolos pueden ser letras, números, símbolos matemáticos e incluso ideogramas y pictogramas.



## Para saber más

Si quieras puedes profundizar en la codificación de caracteres leyendo el siguiente artículo de la Wikipedia.

[Codificación de caracteres.](#)

La forma más habitual de ver escrita una cadena de caracteres es como un literal de cadena. Consiste simplemente en una secuencia de caracteres entre comillas dobles, por ejemplo: "[Ejemplo de cadena de caracteres](#)".

En Java, los literales de cadena son en realidad instancias de la clase `String`, lo cual quiere decir que, por el mero hecho de escribir un literal, se creará una instancia de dicha clase. Esto da mucha flexibilidad, puesto que permite crear cadenas de muchas formas diferentes, pero obviamente consume mucha memoria. La forma más habitual es crear una cadena partiendo de un literal:

```
String cad="Ejemplo de cadena";
```

En este caso, el literal de cadena situado a la derecha del igual es en realidad una instancia de la clase `String`. Al realizar esta asignación hacemos que la variable `cad` se convierta en una referencia al objeto ya creado. Otra forma de crear una cadena es usando el operador `new` y un constructor, como por ejemplo:

```
String cad=new String ("Ejemplo de cadena");
```

Cuando se crean las cadenas de esta forma, se realiza una copia en memoria de la cadena pasada por parámetro. La nueva instancia de la clase `String` hará referencia por tanto a la copia de la cadena, y no al original.

## Reflexiona

Fíjate en el siguiente línea de código, ¿cuantas instancias de la clase `String` ves?

```
String cad="Ejemplo de cadena 1"; cad="Ejemplo de cadena 2"; cad=new String("Ejemplo de cadena 3");
```

[Mostrar retroalimentación](#)

Pues en realidad hay 4 instancias. La primera instancia es la que se crea con el literal de cadena "[Ejemplo de cadena 1](#)". El segundo literal, "[Ejemplo de cadena 2](#)", da lugar a otra instancia diferente a la anterior. El tercer literal, "[Ejemplo de cadena 3](#)", es también nuevamente otra instancia de `String` diferente. Y por último, al crear una nueva instancia de la clase `String` a través del operador `new`, se crea un nuevo objeto `String` copiando para ello el contenido de la

cadena que se le pasa por parámetro, con lo que aquí tenemos la cuarta instancia del objeto **String** en solo una línea.

## 2.1.- Operaciones avanzadas con cadenas de caracteres (I).

¿Qué operaciones puedes hacer con una cadena? Muchas más de las que te imaginas. Empezaremos con la operación más sencilla: la concatenación. La concatenación es la unión de dos cadenas, para formar una sola. En Java es muy sencillo, pues sólo tienes que utilizar el operador de concatenación (signo de suma):

```
String cad = "¡Bien"+"venido!";  
  
System.out.println(cad);
```



Chealer (GNU/GPL)

En la operación anterior se está creando una nueva cadena, resultado de unir dos cadenas: una cadena con el texto "**¡Bien**", y otra cadena con el texto "**venido!**". La segunda línea de código muestra por la salida estándar el resultado de la concatenación. El resultado de su ejecución será que aparecerá el texto "**¡Bienvenido!**" por la pantalla.

Otra forma de usar la concatenación, que ilustra que cada literal de cadena es a su vez una instancia de la clase **String**, es usando el método **concat** del objeto **String**:

```
String cad="¡Bien".concat("venido!");  
  
System.out.printf(cad);
```

Fíjate bien en la expresión anterior, pues genera el mismo resultado que la primera opción y en ambas participan tres instancias de la clase **String**. Una instancia que contiene el texto "**¡Bien**", otra instancia que contiene el texto "**venido!**", y otra que contiene el texto "**¡Bienvenido!**". La tercera cadena se crea nueva al realizar la operación de concatenación, sin que las otras dos hayan desaparecido. Pero no te preocupes por las otras dos cadenas, pues se borrarán de memoria cuando el recolector de basura detecte que ya no se usan.

Fíjate además, que se puede invocar directamente un método de la clase **String**, posponiendo el método al literal de cadena. Esto es una señal de que al escribir un literal de cadena, se crea una instancia del objeto inmutable String.

Pero no solo podemos concatenar una cadena a otra cadena. Gracias al método **toString()** podemos concatenar cadenas con literales numéricos e instancias de otros objetos sin problemas.

El método **toString()** es un método disponible en todas las clases de Java. Su objetivo es simple, permitir la conversión de una instancia de clase en cadena de texto, de forma que se pueda convertir a texto el contenido de la instancia. Lo de convertir, no siempre es posible, hay clases fácilmente convertibles a texto, como es la clase **Integer**, por ejemplo, y otras que no se pueden convertir, y que el resultado de invocar el método **toString()** es información relativa a la instancia.

La gran ventaja de la concatenación es que el método **toString()** se invocará automáticamente, sin que tengamos que especificarlo, por ejemplo:

```
Integer i1=new Integer (1223); // La instancia i1 de la clase Integer contiene el número 1223.  
  
System.out.println("Número: " + i1); // Se mostrará por pantalla el texto "Número: 1223"
```

En el ejemplo anterior, se ha invocado automáticamente **i1.toString()**, para convertir el número a cadena. Esto se realizará para cualquier instancia de clase concatenada, pero cuidado, como se ha dicho antes, no todas las clases se pueden convertir a cadenas.

# Autoevaluación

¿Qué se mostrará como resultado de ejecutar el siguiente código `System.out.println(4+1+"-"+4+1);` ?

- Mostrará la cadena "5-41".
- Mostrará la cadena "41-14".
- Esa operación dará error.

Efectivamente, primero se realiza la suma numérica de 4 y 1, dando 5. Después se concatena 5 a "-" creando una nueva cadena: "5-". Después se concatena la cadena "5-" al número 4, dando lugar a "5-4" y por último se concatena la cadena "5-4" al número 1, dando lugar a la cadena "5-41". Las operaciones se aplican de izquierda a derecha.

Incorrecto. El símbolo es el mismo, la suma, pero se aplican de forma diferente según el contexto, en unos casos significa sumar y en otros concatenación, siempre de izquierda a derecha.

Incorrecto. Los literales enteros se encapsularán en la clase `Integer` cuando sea necesario.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

## 2.1.1.- Operaciones avanzadas con cadenas de caracteres (II).

Vamos a continuar revisando las operaciones que se pueden realizar con cadenas. Como verás las operaciones a realizar se complican un poco a partir de ahora. En todos los ejemplos la variable `cad` contiene la cadena "¡Bienvenido!", como se muestra en las imágenes.

- ✓ `int length()`. Retorna un número entero que contiene la longitud de una cadena, resultado de contar el número de caracteres por la que esta compuesta. Recuerda que un espacio es también un carácter.

String cad="¡Bienvenido!"  
Longitud = 12

Salvador Romero Villegas ([CC BY-NC](#))

- ✓ `char charAt(int pos)`. Retorna el carácter ubicado en la posición pasada por parámetro. El carácter obtenido de dicha posición será almacenado en un tipo de dato `char`. Las posiciones se empiezan a contar desde el 0 (y no desde el 1), y van desde 0 hasta `longitud - 1`. Por ejemplo, el código siguiente mostraría por pantalla el carácter "v":

```
char t = cad.charAt(5);  
  
System.out.println(t);
```

String cad="¡Bienvenido!"  
cad.charAt(0)    cad.charAt(4)    cad.charAt(1)

Salvador Romero Villegas ([CC BY-NC](#))

- ✓ `String substring(int beginIndex, int endIndex)`. Este método permite extraer una subcadena de otra de mayor tamaño. Una cadena compuesta por todos los caracteres existentes entre la posición `beginIndex` y la posición `endIndex - 1`. Por ejemplo, si pusieramos `cad.substring(0,5)` en nuestro programa, sobre la variable `cad` anterior, dicho método devolvería la subcadena "¡Bien" tal y como se muestra en la imagen.

String cad="¡Bienvenido!"  
substring(0,5)    substring(0,11)  
  
String cad="¡Bienvenido!"  
substring(0)

Salvador Romero Villegas ([CC BY-NC](#))

- ✓ `String substring (int beginIndex)`. Cuando al método `substring` solo le proporcionamos un parámetro, extraerá una cadena que comenzará en el carácter con posición `beginIndex` e irá hasta el final de la cadena. En el siguiente ejemplo se mostraría por pantalla la cadena "ienvenido!":

```
String subcad = cad.substring(2);  
  
System.out.println(subcad);
```

Otra operación muy habitual es la conversión de número a cadena y de cadena a número. Imagínate que un usuario introduce su edad. Al recoger la edad desde la interfaz de usuario, capturarás generalmente una cadena, pero, ¿cómo compruebas que la edad es mayor que 0? Para poder realizar esa comprobación tienes que pasar la cadena a número. Empezaremos por ver como se convierte un número a cadena.

Los números generalmente se almacenan en memoria como números binarios, es decir, secuencias de unos y ceros con los que se puede operar (sumar, restar, etc.). No debes confundir los tipos de datos que contienen números (`int`, `short`, `long`, `float` y `double`) con las secuencias de caracteres que representan un número. No es lo mismo 123 que "123", el primero es un número y el segundo es una cadena formada por tres caracteres: '1', '2' y '3'.

Convertir un número a cadena es fácil desde que existe, para todas las clases Java, el método `toString()`. Gracias a ese método podemos hacer cosas como las siguientes:

```
String cad2="Número cinco: " + 5;  
System.out.println(cad2);
```

El resultado del código anterior es que se mostrará por pantalla "Número cinco: 5", y no dará ningún error. Esto es posible gracias a que Java convierte el número 5 a su clase envoltorio (wrapper class) correspondiente (`Integer`, `Float`, `Double`, etc.), y después ejecuta automáticamente el método `toString()` de dicha clase.

## Reflexiona

¿Cuál crees que será el resultado de poner `System.out.println("A"+5f)`? Pruébalo y recuerda: no olvides indicar el tipo de literal (f para los literales de números flotantes, y d para los literales de números dobles), así obtendrás el resultado esperado y no algo diferente.

## 2.1.2.- Operaciones avanzadas con cadenas de caracteres (III).

¿Cómo comprobarías que la cadena "3" es mayor que 0? No puedes comparar directamente una cadena con un número, así que necesitarás aprender cómo convertir cadenas que contienen números a tipos de datos numéricos (`int`, `short`, `long`, `float` o `double`). Esta es una operación habitual en todos los lenguajes de programación, y Java, para este propósito, ofrece los métodos `valueOf`, existentes en todas las clases envoltorio descendientes de la clase `Number`: `Integer`, `Long`, `Short`, `Float` y `Double`.

Veamos un ejemplo de su uso para un número de doble precisión, para el resto de las clases es similar:

```
String c="1234.5678";  
  
double n;  
try {  
  
    n=Double.valueOf(c).doubleValue();  
  
} catch (NumberFormatException e)  
  
{ /* Código a ejecutar si no se puede convertir */ }
```



ITE idITE=109534 ([CC BY-NC](#))

Fijate en el código anterior, en él puedes comprobar cómo la cadena `c` contiene en su interior un número, pero escrito con dígitos numéricos (caracteres). El código escrito está destinado a transformar la cadena en número, usando el método `valueOf`. Este método lanzará la excepción `NumberFormatException` si no consigue convertir el texto a número. En el siguiente archivo, tienes un ejemplo más completo, donde aparecen también ejemplos para las otros tipos numéricos:

### [Ejemplo de conversión de cadena de texto a número.](#)

Seguro que ya te vas familiarizando con este embrollo y encontrarás interesante todas estas operaciones. Ahora te planteamos otro reto: imagina que tienes que mostrar el precio de un producto por pantalla. Generalmente, si un producto vale, por ejemplo 3,3 euros, el precio se debe mostrar como "3,30 €", es decir, se le añade un cero extra al final para mostrar las centésimas. Con lo que sabemos hasta ahora, usando la concatenación en Java, podemos conseguir que una cadena se concatene a un número flotante, pero el resultado no será el esperado. Prueba el siguiente código:

```
float precio=3.3f;  
  
System.out.println("El precio es: "+precio+"€");
```

Si has probado el código anterior, habrás comprobado que el resultado no muestra "3,30 €" sino que muestra "3,3 €". ¿Cómo lo solucionamos? Podríamos dedicar bastantes líneas de código hasta conseguir algo que realmente funcione, pero no es necesario, dado que Java y otros lenguajes de programación (como C), disponen de lo que se denomina formato de cadenas. En Java podemos "formatear" cadenas a través del método estático `format` disponible en el objeto `String`. Este método permite crear una cadena proyectando los argumentos en un formato específico de salida. Lo mejor es verlo con un ejemplo, veamos cuál sería la solución al problema planteado antes:

```
float precio=3.3f;  
  
String salida=String.format ("El precio es: %.2f €", precio);  
  
System.out.println(salida);
```

El formato de salida, también denominado "cadena de formato", es el primer argumento del método `format`. La variable `precio`, situada como segundo argumento, es la variable que se proyectará en la salida siguiendo un formato concreto. Seguro que te preguntarás, ¿qué es "%.2f"? Pues es un especificador de formato, e indica

cómo se deben formatear o proyectar los argumentos que hay después de la cadena de formato en el método `format`.

## Debes conocer

En el [Anexo I](#) de esta Unidad puedes profundizar sobre el método `format` y los especificadores de formato.

## 2.1.3.- Operaciones avanzadas con cadenas de caracteres (IV).

¿Cómo puedo comprobar si dos cadenas son iguales? ¿Qué más operaciones ofrece Java sobre las cadenas? Java ofrece un montón de operaciones más sobre cadenas de caracteres. En la siguiente tabla puedes ver las operaciones más importantes. En todos los ejemplos expuestos, las variables `cad1`, `cad2` y `cad3` son cadenas ya existentes, y la variable `num` es un número entero mayor o igual a cero.

### Métodos importantes de la clase String.

Método.	Descripción
<code>cad1.compareTo(cad2)</code>	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena ( <code>cad1</code> ) es anterior en orden alfabético a la que se pasa por argumento ( <code>cad2</code> ), y un número mayor que cero si la cadena es posterior en orden alfabético.
<code>cad1.equals(cad2)</code>	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación " <code>==</code> ", sino el método <code>equals</code> . Retornará <code>true</code> si son iguales, y <code>false</code> si no lo son.
<code>cad1.compareToIgnoreCase(cad2)</code> <code>cad1.equalsIgnoreCase(cad2)</code>	El método <code>compareToIgnoreCase</code> funciona igual que el método <code>compareTo</code> , pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método <code>equalsIgnoreCase</code> es igual que el método <code>equals</code> pero sin tener en cuenta las minúsculas.
<code>cad1.trim()</code>	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
<code>cad1.toLowerCase()</code>	Genera una copia de la cadena con todos los caracteres a minúscula.
<code>cad1.toUpperCase()</code>	Genera una copia de la cadena con todos los caracteres a mayúsculas.
<code>cad1.indexOf(cad2)</code> <code>cad1.indexOf(cad2, num)</code>	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
<code>cad1.contains(cad2)</code>	Retornará <code>true</code> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <code>false</code> .
<code>cad1.startsWith(cad2)</code>	Retornará <code>true</code> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .



Mass Communication Specialist 3rd Class  
Matthew Jackson, U.S. Navy (Dominio público)

Método.	Descripción
<code>cad1.endsWith(cad2)</code>	Retornará <code>true</code> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.replace(cad2, cad3)</code>	Generará una copia de la cadena <code>cad1</code> , en la que se reemplazarán todas las apariciones de <code>cad2</code> por <code>cad3</code> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzz" generará "xxxx" y no "zxzx".

## Autoevaluación

¿Cuál será el resultado de ejecutar `cad1.replace("l","j").indexOf("ja")` si `cad1` contiene la cadena "hojalata"?

- 2.
- 3.
- 4.
- 1.

Acertaste, esta no era fácil. Enhorabuena.

Incorrecto. Una recomendación, empieza a contar por cero.

No es correcto. Revisa el funcionamiento del método `indexOf`.

Deberías revisar los métodos anteriores. No es correcto.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 2.1.4.- Operaciones avanzadas con cadenas de caracteres (V).

¿Sabes cuál es el principal problema de las cadenas de caracteres? Su alto consumo de memoria. Cuando realizamos un programa que realiza muchísimas operaciones con cadenas, es necesario optimizar el uso de memoria.

En Java, **String** es un objeto inmutable, lo cual significa, entre otras cosas, que cada vez que creamos un **String**, o un literal de **String**, se crea un nuevo objeto que no es modificable. Java proporciona la clase **StringBuilder**, la cual es un **mutable**, y permite una mayor optimización de la memoria. También existe la clase **StringBuffer**, pero consume mayores recursos al estar pensada para aplicaciones **multi-hilo**, por lo que en nuestro caso nos centraremos en la primera.

Pero, ¿en qué se diferencian **StringBuilder** de la clase **String**? Pues básicamente en que la clase **StringBuilder** permite modificar la cadena que contiene, mientras que la clase **String** no. Como ya se dijo antes, al realizar operaciones complejas se crea una nueva instancia de la clase **String**.

Veamos un pequeño ejemplo de uso de esta clase. En el ejemplo que vas a ver, se parte de una cadena con errores, que modificaremos para ir haciéndola legible. Lo primero que tenemos que hacer es crear la instancia de esta clase. Se puede inicializar de muchas formas, por ejemplo, partiendo de un literal de cadena:

```
StringBuilder strb=new StringBuilder ("Hoal Muuundo");
```

Y ahora, usando los métodos **append** (insertar al final), **insert** (insertar una cadena o carácter en una posición específica), **delete** (eliminar los caracteres que hay entre dos posiciones) y **replace** (reemplazar los caracteres que hay entre dos posiciones por otros diferentes), rectificaremos la cadena anterior y la haremos correcta:

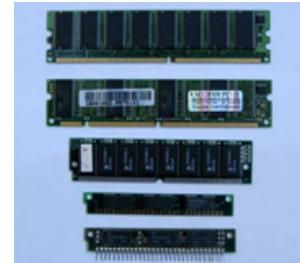
1. **strb.delete(6,8);** Eliminamos las 'uu' que sobran en la cadena. La primera 'u' que sobra está en la posición 6 (no olvides contar el espacio), y la última 'u' a eliminar está en la posición 7. Para eliminar dichos caracteres de forma correcta hay que pasar como primer argumento la posición 6 (posición inicial) y como segundo argumento la posición 8 (posición contigua al último carácter a eliminar), dado que la posición final no indica el último carácter a eliminar, sino el carácter justo posterior al último que hay que eliminar (igual que ocurría con el método **substring**).
2. **strb.append ("!");** Añadimos al final de la cadena el símbolo de cierre de exclamación.
3. **strb.insert (0,"i");** Insertamos en la posición 0, el símbolo de apertura de exclamación.
4. **strb.replace (3,5,"la");** Reemplazamos los caracteres 'al' situados entre la posición inicial 3 y la posición final 4, por la cadena 'la'. En este método ocurre igual que en los métodos **delete** y **substring**, en vez de indicar como posición final la posición 4, se debe indicar justo la posición contigua, es decir 5.

**StringBuilder** contiene muchos métodos de la clase **String** (**charAt**, **indexOf**, **length**, **substring**, **replace**, etc.), pero no todos, pues son clases diferentes con funcionalidades realmente diferentes.

### Debes conocer

En la siguiente página puedes encontrar más información (en inglés) sobre como utilizar la clase **StringBuilder**.

[Uso de la clase StringBuilder.](#)



ITE. Juan Manuel Rubio Marauri  
idITE=133011 (CC BY-NC)

### Autoevaluación

**Rotar una cadena es poner simplemente el primer carácter al final, y retroceder el resto una posición. Después de unas cuantas rotaciones la cadena queda igual. ¿Cuál de las siguientes expresiones serviría para hacer una rotación (rotar solo una posición)?**

- `stb.delete (0,1); strb.append(stb.charAt(0));`
- `strb.append(strb.charAt(0));strb.delete(0, 1);`
- `strb.append(strb.charAt(0));strb.delete(0);`
- `strb.append(strb.charAt(1));strb.delete(1,2);`

No es correcto. No puedes eliminar el primer carácter insertarlo al final.

Correcto, muy bien.

Incorrecto. El método `delete` necesita dos parámetros.

Incorrecto. La primera posición no es la 1, sino la 0.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 2.2.- Expresiones regulares (I).

¿Tienen algo en común todos los números de DNI y de NIE? ¿Podrías hacer un programa que verificara si un DNI o un NIE es correcto? Seguro que sí. Si te fijas, los números de DNI y los de NIE tienen una estructura fija: X1234567Z (en el caso del NIE) y 1234567Z (en el caso del DNI). Ambos siguen un patrón que podría describirse como: una letra inicial opcional (solo presente en los NIE), seguida de una secuencia numérica y finalizando con otra letra. ¿Fácil no?



ITE idITE=111041 (CC BY-NC)

Pues esta es la función de las expresiones regulares: **permitir comprobar si una cadena sigue o no un patrón preestablecido**. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares, y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial. Por ejemplo "[01]" es una expresión regular que permite comprobar si una cadena conforma un número binario. Veamos cuáles son las reglas generales para construir una expresión regular:

- ✓ Podemos indicar que una cadena contiene un conjunto de símbolos fijo, simplemente poniendo dichos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán un carácter de escape como veremos más adelante. Por ejemplo, el patrón "aaa" admitirá cadenas que contengan tres a's.
- ✓ "[xyz]". Entre corchetes podemos indicar opcionalidad. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes. Por ejemplo, la expresión regular "aaa[xy]" admitirá como válidas las cadenas "aaax" y la cadena "aaay". **Los corchetes representan una posición de la cadena que puede tomar uno de varios valores posibles.**
- ✓ "[a-z]" "[A-Z]" "[a-zA-Z]". Usando el guion y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final. Es importante que sepas que se diferencia entre letras mayúsculas y minúsculas, no son iguales de cara a las expresiones regulares.
- ✓ "[0-9]". Y nuevamente, usando un guion, podemos indicar que se permite la presencia de un dígito numérico entre 0 y 9, cualquiera de ellos, pero solo uno.

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón. Veamos ahora como indicar repeticiones:

- ✓ "a?". Usaremos el interrogante para indicar que un símbolo puede aparecer una vez o ninguna. De esta forma la letra "a" podrá aparecer una vez o simplemente no aparecer.
- ✓ "a\*". Usaremos el asterisco para indicar que un símbolo puede aparecer una vez o muchas veces, pero también ninguna. Cadenas válidas para esta expresión regular serían "aa", "aaa" o "aaaaaaaa".
- ✓ "a+". Usaremos el símbolo de suma para indicar que otro símbolo debe aparecer al menos una vez, pudiendo repetirse cuantas veces quiera.
- ✓ "a{1,4}" . Usando las llaves, podemos indicar el número mínimo y máximo de veces que un símbolo podrá repetirse. El primer número del ejemplo es el número 1, y quiere decir que la letra "a" debe aparecer al menos una vez. El segundo número, 4, indica que como máximo puede repetirse cuatro veces.
- ✓ "a{2,}". También es posible indicar solo el número mínimo de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
- ✓ "a{5}". A diferencia de la forma anterior, si solo escribimos un número entre llaves, sin poner la coma detrás, significará que el símbolo debe aparecer un número exacto de veces. En este caso, la "a" debe aparecer exactamente 5 veces.
- ✓ "[a-zA-Z]{1,4}[0-9]+". Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

### Ejercicio resuelto

Con lo visto hasta ahora ya es posible construir una expresión regular capaz de verificar si una cadena contiene un DNI o un NIE, ¿serías capaz de averiguar cuál es dicha expresión regular?

[Mostrar retroalimentación](#)

La expresión regular que permite verificar si una cadena contiene un DNI o un NIE es la siguiente: "[XYxy]?[0-9]{1,9}[A-Za-z]"; aunque no es la única solución.

## 2.2.1.- Expresiones regulares (II).

¿Y cómo uso las expresiones regulares en un programa? Pues de una forma sencilla. Para su uso, Java ofrece las clases **Pattern** y **Matcher** contenidas en el paquete **java.util.regex.\***. La clase **Pattern** se utiliza para procesar la expresión regular y "compilarla", lo cual significa verificar que es correcta y dejarla lista para su utilización. La clase **Matcher** sirve para comprobar si una cadena cualquiera sigue o no un patrón. Veamoslo con un ejemplo:



ITE idITE=110172 (CC BY-NC)

```
Pattern p=Pattern.compile("[01]+");
Matcher m=p.matcher("00001010");
if (m.matches()) System.out.println("Si, contiene el patrón");
else System.out.println("No, no contiene el patrón");
```

En el ejemplo, el método estático **compile** de la clase **Pattern** permite crear un patrón, dicho método compila la expresión regular pasada por parámetro y genera una instancia de **Pattern** (**p** en el ejemplo). El patrón **p** podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método **matcher**, el cual combina el patrón con la cadena de entrada y genera una instancia de la clase **Matcher** (**m** en el ejemplo). La clase **Matcher** contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- ✓ **m.matches()**. Devolverá **true** si toda la cadena (de principio a fin) encaja con el patrón o **false** en caso contrario.
- ✓ **m.lookingAt()**. Devolverá **true** si el patrón se ha encontrado al principio de la cadena. A diferencia del método **matches()**, la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.
- ✓ **m.find()**. Devolverá **true** si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y **false** en caso contrario, pudiendo tener más de una coincidencia. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos **m.start()** y **m.end()**, para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método **find()**, irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método **find()**, para que vuelva a comenzar por la primera coincidencia, invocando el método **m.reset()**.

Veamos algunas construcciones adicionales que pueden ayudarnos a especificar expresiones regulares más complejas:

- ✓ "[^abc]". El símbolo "^", cuando se pone justo detrás del corchete de apertura, significa "negación". La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes. En este caso, cualquier símbolo diferente de "a", "b" o "c".
- ✓ "[01]+\$". Cuando el símbolo "^" aparece al comienzo de la expresión regular, permite indicar comienzo de línea o de entrada. El símbolo "\$" permite indicar fin de línea o fin de entrada. Usándolos podemos verificar que una línea completa (de principio a fin) encaje con la expresión regular, es muy útil cuando se trabaja en modo multilínea y con el método **find()**.
- ✓ ". ". El punto simboliza cualquier carácter.
- ✓ "\d". Un dígito numérico. Equivale a "[0-9]".
- ✓ "\p". Cualquier cosa excepto un dígito numérico. Equivale a "[^0-9]".
- ✓ "\s". Un espacio en blanco (incluye tabulaciones, saltos de línea y otras formas de espacio).
- ✓ "\S". Cualquier cosa excepto un espacio en blanco.
- ✓ "\w". Cualquier carácter que podrías encontrar en una palabra. Equivale a "[a-zA-Z\_0-9]".

### Autoevaluación

¿En cuáles de las siguientes opciones se cumple el patrón "A.\d+"?

- "GA-99" si utilizamos el método **find**.

- "GAX99" si utilizamos el método `lookingAt`.

- "AX99-" si utilizamos el método `matches`.

- "A99" si utilizamos el método `matches`.

[Mostrar retroalimentación](#)

## Solución

1. Correcto
2. Incorrecto
3. Incorrecto
4. Correcto

## 2.2.2.- Expresiones regulares (III).

¿Te resultan difíciles las expresiones regulares? Al principio siempre lo son, pero no te preocupes. Hasta ahora has visto como las expresiones regulares permiten verificar datos de entrada, permitiendo comprobar si un dato indicado sigue el formato esperado: que un DNI tenga el formato esperado, que un email sea un email y no otra cosa, etc. Pero ahora vamos a dar una vuelta de tuerca adicional.

Los paréntesis, de los cuales no hemos hablado hasta ahora, tienen un significado especial, permiten indicar repeticiones para un conjunto de símbolos, por ejemplo: "#[01]{2,3}". En el ejemplo anterior, la expresión "#[01]" admitiría cadenas como "#0" o "#1", pero al ponerlo entre paréntesis e indicar los contadores de repetición, lo que estamos diciendo es que la misma secuencia se tiene que repetir entre dos y tres veces, con lo que las cadenas que admitiría serían del estilo a: "#0#1" o "#0#1#0".



Pero los paréntesis tienen una función adicional, y es la de permitir definir grupos. Un grupo comienza cuando se abre un paréntesis y termina cuando se cierra el paréntesis. Los grupos permiten acceder de forma cómoda a las diferentes partes de una cadena cuando esta coincide con una expresión regular. Lo mejor es verlo con un ejemplo (seguro que te resultará familiar):

```
Pattern p=Pattern.compile("[XY]?([0-9]{1,9})([A-Za-z])");
Matcher m=p.matcher("X123456789Z Y00110011M 999999T");
while (m.find())
{
    System.out.println("Letra inicial (opcional):"+m.group(1));
    System.out.println("Número:"+m.group(2));
    System.out.println("Letra NIF:"+m.group(3));
}
```

Usando los grupos, podemos obtener por separado el texto contenido en cada uno de los grupos. En el ejemplo anterior, en el patrón hay tres grupos: uno para la letra inicial (grupo 1), otro para el número del DNI o NIE (grupo 2), y otro para la letra final o letra NIF (grupo 3). Al ponerlo en grupos, usando el método `group()`, podemos extraer la información de cada grupo y usarla a nuestra conveniencia.

Ten en cuenta que el primer grupo es el 1, y no el 0. Si pones `m.group(0)` obtendrás una cadena con toda la ocurrencia o coincidencia del patrón en la cadena, es decir, obtendrás la secuencia entera de símbolos que coincide con el patrón.

En el ejemplo anterior se usa el método `find`, éste buscará una a una, cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devolverá `true` si ha encontrado una ocurrencia. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más, y retornará `false`, saliendo del bucle. Esta construcción `while` es muy típica para este tipo de métodos y para las iteraciones, que veremos más adelante.

Lo último importante de las expresiones regulares que debes conocer son las secuencias de escape. Cuando en una expresión regular necesitamos especificar que lo que tiene que haber en la cadena es un paréntesis, una llave, o un corchete, tenemos que usar una secuencia de escape, dado que esos símbolos tienen un significado especial en los patrones. Para ello, simplemente antepondremos "\\" al símbolo. Por ejemplo, "\\(" significará que debe haber un paréntesis en la cadena y se omitirá el significado especial del paréntesis. Lo mismo ocurre con "\\[", "\\]", "\\\"", etc. Lo mismo para el significado especial del punto, éste, tiene un significado especial (¿Lo recuerdas del apartado anterior?) salvo que se ponga "\\.", que pasará a significar "un punto" en vez de "cualquier carácter". **La excepción son las comillas, que se pondrán con una sola barra: "\".**

## Para saber más

Con lo estudiado hasta ahora, ya puedes utilizar las expresiones regulares y sacarles partido casi que al máximo. Pero las expresiones regulares son un campo muy amplio, por lo que es muy aconsejable que amplíes tus conocimientos con el siguiente enlace:

[Tutorial de expresiones regulares en Java \(en inglés\).](#)

## Debes conocer

Existen herramientas online disponibles para evaluar expresiones regulares en diferentes lenguajes, entre ellas Java.

En el siguiente enlace puedes acceder a un ejemplo de los muchos que existen en la red.

[Freeformatter.com](#)

### 3.- Creación de arrays.

#### Caso práctico

Después del susto inicial, al recibir el archivo con el formato de pedido, **Ana** se puso a pensar. Vio que lo más adecuado era procesar el archivo de pedido línea a línea, viendo con qué corresponde cada línea a través de expresiones regulares. Aún no le preocupa el procesamiento del fichero, sino más bien cómo procesar los datos leídos del mismo.

Para identificar si hay un inicio o fin de sección dentro del pedido, así como el inicio o fin del listado de artículos, ha decidido usar la siguiente expresión regular:

```
"^##[ ]*(FIN){0,1}[ ]*(PEDIDO|ARTICULOS)[ ]*##$".
```



ITE idITE=109713 ([CC BY-NC](#))

Y para identificar cada dato del pedido (nombre, dirección, etc.) va a utilizar la siguiente expresión regular:

```
"^(.+):.*\\{(.*)\\}$"
```

Al usar grupos le permitirá separar el nombre del campo (dirección por ejemplo) de su valor (dirección concreta). La expresión regular le ha costado mucho trabajo y ha tenido que pedir ayuda, pero después, una vez que la ha conseguido, se ha dado cuenta de que ha sido relativamente fácil.

Ana ha decidido de todas formas verificar sus expresiones regulares a través de alguna herramienta online. Para ello, se ha puesto a investigar sobre algunas de ellas y en principio utilizará **freeformatter**. Para ello, tomando los datos de ejemplo del fichero de pedidos, podrá comprobar que sus expresiones regulares están bien formadas.

Ya tiene parte del trabajo hecho. Como comentamos arriba, aún no le preocupa el procesado del fichero sino la parte de artículos. Básicamente tiene dudas sobre donde almacenar la información procesada, en especial, la lista de artículos.

¿Y dónde almacenarías tú la lista de artículos del pedido? Una de las soluciones es usar arrays, puede que sea justo lo que necesita Ana, pero puede que no. Todos los lenguajes de programación permiten el uso de arrays, veamos como son en Java.



Salvador Romero Villegas. ([CC BY-NC](#))

**Los arrays permiten almacenar una colección de objetos o datos del mismo tipo.** Son muy útiles y su utilización es muy simple:

- ✓ **Declaración del array.** La declaración de un array consiste en decir "esto es un array" y sigue la siguiente estructura: "tipo[] nombre;". El tipo será un tipo de variable o una clase ya existente, de la cual se quieran almacenar varias unidades.
- ✓ **Creación del array.** La creación de un array consiste en decir el tamaño que tendrá el array, es decir, el número de elementos que contendrá, y se pone de la siguiente forma: "nombre=new tipo[dimensión]", donde dimensión es un número entero positivo que indicará el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

Veamos un ejemplo de su uso:

```
int[] n; // Declaración del array.
```

```
n = new int[10]; //Creación del array reservando para él un espacio en memoria.
```

```
int[] m=new int[10]; // Declaración y creación en un mismo lugar.
```

Una vez hecho esto, ya podemos almacenar valores en cada una de las posiciones del array, usando corchetes e indicando en su interior la posición en la que queremos leer o escribir, teniendo en cuenta que la primera posición es la cero y la última el tamaño del array menos uno. En el ejemplo anterior, la primera posición sería la 0 y la última sería la 9.

## Autoevaluación

¿Cuáles de las siguientes opciones permitiría almacenar más de 50 números decimales?

`int[] numeros; numeros=new int[51];`

`int[] numeros; numeros=new float[52];`

`double[] numeros; numeros=new double[53];`

`float[] numeros=new float[54];`

[Mostrar retroalimentación](#)

## Solución

1. Incorrecto
2. Incorrecto
3. Correcto
4. Correcto

## 3.1.- Uso de arrays unidimensionales.

Ya sabes declarar y crear de arrays, pero, ¿cómo y cuando se usan? Pues existen tres ámbitos principalmente donde se pueden usar, y los tres son muy importantes: modificación de una posición del array, acceso a una posición del array y paso de parámetros.

La modificación de una posición del array se realiza con una simple asignación. Simplemente se especifica entre corchetes la posición a modificar después del nombre del array. Veamoslo con un simple ejemplo:



Stockbyte DVD-CD Num. V43 ([CC BY-NC](#))

```
int[] Numeros=new int[3]; // Array de 3 números (posiciones del 0 al 2).  
  
Numeros[0]=99; // Primera posición del array.  
  
Numeros[1]=120; // Segunda posición del array.  
  
Numeros[2]=33; // Tercera y última posición del array.
```

El acceso a un valor ya existente dentro de una posición del array se consigue de forma similar, simplemente poniendo el nombre del array y la posición a la cual se quiere acceder entre corchetes:

```
int suma=Numeros[0] + Numeros[1] + Numeros[2];
```

Para nuestra comodidad, los arrays, como objetos que son en Java, disponen de una propiedad pública muy útil. La propiedad `length` nos permite saber el tamaño de cualquier array, lo cual es especialmente útil en métodos que tienen como argumento un array.

```
System.out.println("Longitud del array: "+Numeros.length);
```

El tercer uso principal de los arrays, como se dijo antes, es en el paso de parámetros. Para pasar como argumento un array a una función o método, esta debe tener en su definición un parámetro declarado como array. Esto es simplemente que uno de los parámetros de la función sea un array. Veamos un ejemplo:

```
int sumaarray (int[] j) {  
    int suma=0;  
    for (int i=0; i<j.length;i++)  
        suma=suma+j[i];  
    return suma;  
}
```

En el método anterior se pasa como argumento un array numérico, sobre el cual se calcula la suma de todos los números que contiene. Es un uso típico de los arrays, fíjate que especificar que un argumento es un array es igual que declarar un array, sin la creación del mismo. Para pasar como argumento un array a una función, simplemente se pone el nombre del array:

```
int suma=sumaarray (Numeros);
```

En Java las variables se pasan por copia a los métodos, esto quiere decir que cuando se pasa una variable a un método, y se realiza una modificación de su valor en dicho método, el valor de la variable en el método desde el que se ha realizado la invocación no se modifica. Pero cuidado, eso no pasa con los arrays. Cuando dicha modificación se realiza en un array, es decir, se cambia el valor de uno de los elementos del array, si que cambia su valor de forma definitiva. Veamos un ejemplo que ilustra ambas cosas:

```
public static void main(String[] args) {  
  
    int j=0; int[] i=new int(1); i[0]=0;  
  
    modificaArray(j,i);  
  
    System.out.println(j+"-"+i[0]); /* Mostrará por pantalla "0-1", puesto que el contenido del array es  
    modificado en la función, y aunque la variable j también se modifica, se modifica una copia de l  
    dejando el original intacto */  
  
}  
  
int modificaArray(int j, int[] i) {  
  
    j++; i[0]++; /* Modificación de los valores de la variable, solo afectará al array, no a j */  
  
}
```

## 3.2.- Inicialización.

Rellenar un array, para su primera utilización, es una tarea muy habitual que puede ser rápidamente simplificada. Vamos a explorar dos sistemas que nos van a permitir inicializar un array de forma cómoda y rápida.

En primer lugar, una forma habitual de crear un array y rellenarlo es simplemente a través de un método que lleve a cabo la creación y el relleno del array. Esto es sencillo desde que podemos hacer que un método retorne un array simplemente indicando en la declaración que el valor retornado es tipo[], donde tipo de dato primitivo (`int`, `short`, `float`,...) o una clase existente (`String` por ejemplo). Veamos un ejemplo:

```
static int[] ArrayConNumerosConsecutivos (int totalNumeros) {  
    int[] r=new int[totalNumeros];  
  
    for (int i=0;i<totalNumeros;i++) r[i]=i;  
  
    return r;  
}
```



ITE. Francisco Javier Martínez  
Adrados ([CC BY-NC](#))

En el ejemplo anterior se crea un array con una serie de números consecutivos, empezando por el cero, ¿sencillo no? Este uso suele ahorrar bastantes líneas de código en tareas repetitivas. Otra forma de inicializar los arrays, cuando el número de elementos es fijo y sabido a priori, es indicando entre llaves el listado de valores que tiene el array. En el siguiente ejemplo puedes ver la inicialización de un array de tres números, y la inicialización de un array con tres cadenas de texto:

```
int[] array = {10, 20, 30};  
  
String[] diasSemana= {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"};
```

Pero cuidado, la inicialización solo se puede usar en ciertos casos. La inicialización anterior funciona cuando se trata de un tipo de dato primitivo (`int`, `short`, `float`, `double`, etc.) o un `String`, y algunos pocos casos más, pero no funcionará para cualquier objeto.

Cuando se trata de un array de objetos, la inicialización del mismo es un poco más liosa, dado que el valor inicial de los elementos del array de objetos será `null`, o lo que es lo mismo, crear un array de objetos no significa que se han creado las instancias de los objetos. Hay que crear, para cada posición del array, el objeto del tipo correspondiente con el operador `new`. Veamos un ejemplo con la clase `StringBuilder`. En el siguiente ejemplo solo aparecerá `null` por pantalla:

```
StringBuilder[] j=new StringBuilder[10];  
  
for (int i=0; i<j.length;i++) System.out.println("Valor" +i +"="+j[i]); // Imprimirá null para los 10 valores.
```

Para solucionar este problema podemos optar por lo siguiente, crear para cada posición del array una instancia del objeto:

```
StringBuilder[] j=new StringBuilder[10];  
  
for (int i=0; i<j.length;i++) j[i]=new StringBuilder("cadena "+i);
```

## Reflexiona

Para acceder a una propiedad o a un método cuando los elementos del array son objetos, puedes usar la notación de punto detrás de los corchetes, por ejemplo: `diassemana[0].length`. Fijate bien en el array `diassemana` anterior y piensa en lo que se mostraría por pantalla con el siguiente código:

```
System.out.println(diassemana[0].substring(0,2));
```

## Autoevaluación

¿Cuál es el valor de la posición 3 del siguiente array: `String[] m=new String[10]`?

- `null`.
- Una cadena vacía.
- Daría error y no se podría compilar.

Efectivamente, los objetos no se han creado todavía.

Incorrecto, recuerda que `String` es al fin y al cabo un objeto.

Incorrecto, si se puede compilar. Revisa el apartado.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

## 4.- Arrays multidimensionales.

### Caso práctico



Ministerio de Educación ([CC BY-NC](#))

Ana sigue dándole vueltas a como almacenar los diferentes datos del pedido, y sobre todo, sobre como procesar los artículos, dado que los artículos del pedido pueden ser más de uno. Ha pensado en crear primero una clase para almacenar los datos del pedido, llamada **Pedido**, donde cada dato del pedido sea un atributo de la clase. Pero claro, los artículos son más de uno y no puede saber cuantos atributos necesitará.

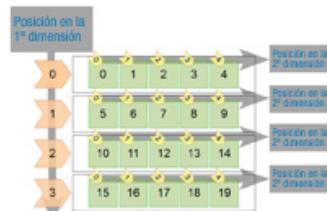
Para almacenar los artículos ha pensado en crear una clase llamada **Artículo** donde se meterían todos los datos de cada artículo, y después en la clase **Pedido** poner un array de artículos. Pero no lo tiene claro y ha decidido preguntar a su tutora, María.

¿Qué estructura de datos utilizarías para almacenar los \_\_\_\_\_ píxeles de una imagen digital? Normalmente las imágenes son cuadradas así que una de las estructuras más adecuadas es la \_\_\_\_\_ matriz. En la matriz cada valor podría ser el color de cada píxel. Pero, ¿qué es una matriz a nivel de programación? Pues es un array con dos dimensiones, o lo que es lo mismo, un array cuyos elementos son arrays de números.

Los arrays multidimensionales están en todos los lenguajes de programación actuales, y obviamente también en Java. La forma de crear un array de dos dimensiones en Java es la siguiente:

```
int[][][] a2d=new int[4][5];
```

El código anterior creará un array de dos dimensiones, o lo que es lo mismo, creará un array que contendrá 4 arrays de 5 números cada uno. Veámoslo con un ejemplo gráfico:



Salvador Romero Villegas ([CC BY-NC](#))

Al igual que con los arrays de una sola dimensión, los arrays multidimensionales deben declararse y crearse. Podremos hacer arrays multidimensionales de todas las dimensiones que queramos y de cualquier tipo. En ellos todos los elementos del array serán del mismo tipo, como en el caso de los arrays de una sola dimensión. La declaración comenzará especificando el tipo o la clase de los elementos que forman el array, después pondremos tantos corchetes como dimensiones tenga el array y por último el nombre del array, por ejemplo:

```
int [[[ ]]] arrayde3dim;
```

La creación es igualmente usando el operador **new**, seguido del tipo y los corchetes, en los cuales se especifica el tamaño de cada dimensión:

```
arrayde3dim=new int[2][3][4];
```

Todo esto, como ya has visto en un ejemplo anterior, se puede escribir en una única sentencia.

## Autoevaluación

Completa con los números que faltan:

```
int[][][] k=new int[10][11][12];
```

El array anterior es de  dimensiones, y tiene un total de  números enteros.

## 4.1.- Uso de arrays multidimensionales.

¿Y en que se diferencia el uso de un array multidimensional con respecto a uno de una única dimensión? Pues en muy poco la verdad. Continuaremos con el ejemplo del apartado anterior:

```
int[][] a2d=new int[4][5];
```

Para acceder a cada uno de los elementos del array anterior, habrá que indicar su posición en las dos dimensiones, teniendo en cuenta que los índices de cada una de las dimensiones empieza a numerarse en 0 y que la última posición es el tamaño de la dimensión en cuestión menos 1.

Puedes asignar un valor a una posición concreta dentro del array, indicando la posición en cada una de las dimensiones entre corchetes:

```
a2d[0][0]=3;
```

Y como es de imaginar, puedes usar un valor almacenado en una posición del array multidimensional simplemente poniendo el nombre del array y los índices del elemento al que deseas acceder entre corchetes, para cada una de las dimensiones del array. Por ejemplo:

```
int suma=a2d[0][0]+a2d[0][1]+a2d[0][2]+a2d[0][3]+a2d[0][4];
```

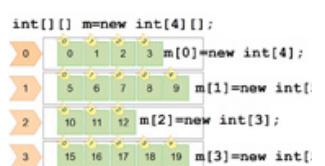
Como imaginarás, los arrays multidimensionales pueden también ser pasados como parámetros a los métodos, simplemente escribiendo la declaración del array en los argumentos del método, de forma similar a como se realizaba para arrays de una dimensión. Por ejemplo:

```
static int sumaarray2d(int[][] a2d) {  
    int suma = 0;  
  
    for (int i1 = 0; i1 < a2d.length; i1++)  
  
        for (int i2 = 0; i2 < a2d[i1].length; i2++) suma += a2d[i1][i2];  
  
    return suma;  
}
```

Del código anterior, fíjate especialmente en el uso del atributo `length` (que nos permite obtener el tamaño de un array). Aplicado directamente sobre el array nos permite saber el tamaño de la primera dimensión (`a2d.length`). Como los arrays multidimensionales son arrays que tienen como elementos arrays (excepto el último nivel que ya será del tipo concreto almacenado), para saber el tamaño de una dimensión superior tenemos que poner el o los índices entre corchetes seguidos de `length` (`a2d[i1].length`).

Para saber al tamaño de una segunda dimensión (dentro de una función por ejemplo) hay que hacerlo así y puede resultar un poco engorroso, pero gracias a esto podemos tener arrays multidimensionales irregulares.

Una matriz es un ejemplo de array multidimensional regular, ¿por qué? Pues porque es un array que contiene arrays de números todos del mismo tamaño. Cuando esto no es así, es decir, cuando los arrays de la segunda dimensión son de diferente tamaño entre sí, se puede decir que es un array multidimensional irregular. En Java se puede crear un array irregular de forma relativamente fácil, veamos un ejemplo para dos dimensiones.



- ✓ **Declaramos y creamos el array pero sin especificar la segunda dimensión.** Lo que estamos haciendo en realidad es crear simplemente un array que contendrá arrays, sin decir como son de grandes los arrays de la siguiente dimensión: `int[][] irregular=new int[3][];`
- ✓ **Después creamos cada uno de los arrays unidimensionales** (del tamaño que queramos) y lo asignamos a la posición correspondiente del array anterior: `irregular[0]=new int[7]; irregular[1]=new int[15]; irregular[2]=new int[9];`

## Recomendación

Cuando uses arrays irregulares, por seguridad, es conveniente que verifiques siempre que el array no sea `null` en segundas dimensiones, y que la longitud sea la esperada antes de acceder a los datos:

```
if (irregular[1]!=null && irregular[1].length>10) {...}
```

## 4.2.- Inicialización de arrays multidimensionales.

¿En qué se diferencia la inicialización de arrays unidimensionales de arrays multidimensionales? En muy poco. La inicialización de los arrays multidimensionales es igual que la de los arrays unidimensionales.

Para que una función retorne un array multidimensional, se hace igual que en arrays unidimensionales. Simplemente hay que poner el tipo de dato seguido del número de corchetes correspondiente, según el número de dimensiones del array. Eso claro, hay que ponerlo en la definición del método:

```
int[][][] inicializarArray (int n, int m)

{
    int[][][] ret=new int[n][m];

    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            ret[i][j]=n*m;

    return ret;
}
```

También se puede inicializar un array multidimensional usando las llaves, poniendo después de la declaración del array un símbolo de igual, y encerrado entre llaves los diferentes valores del array separados por comas, con la salvedad de que hay que poner unas llaves nuevas cada vez que haya que poner los datos de una nueva dimensión, lo mejor es verlo con un ejemplo:

```
int[][][] a2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};

int[][][] a3d={{{{0,1},{2,3}},{{0,1},{2,3}}}};
```

El primer array anterior sería un array de 4 por 3 y el siguiente sería un array de 2x2x2. Como puedes observar esta notación a partir de 3 dimensiones ya es muy liosa y normalmente no se usa. Utilizando esta notación también podemos inicializar rápidamente arrays irregulares, simplemente poniendo separados por comas los elementos que tiene cada dimensión:

```
int[][][] i2d={{0,1},{0,1,2},{0,1,2,3},{0,1,2,3,4},{0,1,2,3,4,5}};

int[][][] i3d={ {{0,1},{0,2}} , {{0,1,3}} , {{0,3,4},{0,1,5}} };
```

Es posible que en algunos libros y en Internet se refieran a los arrays usando otra terminología. A los arrays unidimensionales es común llamarlos también **arreglos** o **vectores**, a los arrays de dos dimensiones es común llamarlos directamente **matrices**, y a los arrays de más de dos dimensiones es posible que los veas escritos como **matrices multidimensionales**. Sea como sea, lo más normal en la jerga informática es llamarlos arrays, término que procede del inglés.

### Autoevaluación

Dado el siguiente array: `int[][][] i3d={{ {0,1},{0,2}}, {{0,1,3}},{{0,3,4},{0,1,5}}};`

¿Cuál es el valor de la posición [1][1][2]?

- 3
- 4
- 5
- Ninguno de los anteriores.

No es correcto. No es lo que parece, algo se te ha pasado por alto.

No es correcto, deberías analizar con mayor detenimiento el array.

Incorrecto, revisa de nuevo el array.

Muy bien, efectivamente esa posición no existe y si intentas acceder a ella sin verificarlo obtendrías una excepción tipo `ArrayIndexOutOfBoundsException` (excepción de indice de array fuera de los límites).

## Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

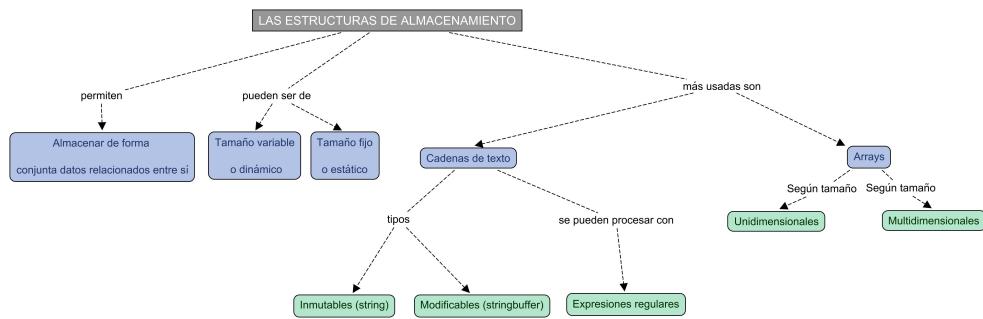
## Para saber más

Las matrices tienen asociadas un amplio abanico de operaciones (transposición, suma, producto, etc.). En la siguiente página tienes ejemplos de como realizar algunas de estas operaciones, usando por supuesto arrays:

[Operaciones básicas con matrices.](#)

## 5.- Conclusiones.

A lo largo de esta unidad hemos trabajado con las primeras estructuras de datos compuestas, es decir, aquellas que están formadas por datos primitivos u otros datos compuestos. Como hemos podido comprobar, estas estructuras son necesarias para almacenar colecciones de datos. Entre las estructuras trabajadas en esta unidad se encuentran las cadenas de texto y los arrays, que tienen como particularidad de que son estructuras estáticas, es decir, su tamaño se define en tiempo de compilación y no puede variar durante la ejecución de la aplicación. Además, se han introducido las expresiones regulares como una de las herramientas más potentes para la validación de cadenas de texto.



En próximas unidades trabajaremos con estructuras de datos dinámicas. Pero antes, en la unidad de trabajo 6, introduciremos y trataremos de que queden claros dos de los conceptos más importantes de la POO, la herencia y el polimorfismo, que además están íntimamente relacionados con las unidades de trabajo restantes. Además, trabajaremos con otros conceptos avanzados relacionados con la POO.

# Anexo I.- Formateado de cadenas en Java.

## Sintaxis de las cadenas de formato y uso del método format.

En Java, el método estático `format` de la clase `String` permite formatear los datos que se muestran al usuario o la usuaria de la aplicación. El método `format` tiene los siguientes argumentos:

- ✓ Cadena de formato. Cadena que especifica cómo será el formato de salida, en ella se mezclará texto normal con especificadores de formato, que indicarán cómo se debe formatear los datos.
- ✓ Lista de argumentos. Variables que contienen los datos cuyos datos se formatearán. Tiene que haber tantos argumentos como especificadores de formato haya en la cadena de formato.

Los especificadores de formato comienzan siempre por "%", es lo que se denomina un carácter de escape (carácter que sirve para indicar que lo que hay a continuación no es texto normal, sino algo especial). El especificador de formato debe llevar como mínimo el símbolo "%" y un carácter que indica la conversión a realizar, por ejemplo "%d".

La conversión se indica con un simple carácter, y señala al método `format` cómo debe ser formateado el argumento. Dependiendo del tipo de dato podemos usar unas conversiones u otras. Veamos las conversiones más utilizadas:

## Listado de conversiones más utilizada y ejemplos.

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo	Resultado del ejemplo
Valor lógico o booleano.	"%b" O "%B"	Boolean (cuando se usan otros tipos de datos siempre lo formateará escribiendo true).	<pre>boolean b=true;  String d= String.format("Resultado: %b", b);  System.out.println (d);</pre>	Resultado: true
Cadena de caracteres.	"%s" O "%\$"	Cualquiera, se convertirá el objeto a cadena si es posible (invocando el método <code>toString</code> ).	<pre>String cad="hola mundo";  String d= String.format("Resultado: %s", cad);  System.out.println (d);</pre>	Resultado: hola mundo
Entero decimal	"%d"	Un tipo de dato entero.	<pre>int i=10;  String d= String.format("Resultado: %d", i);  System.out.println (d);</pre>	Resultado: 10

Tipo de conversión	Especificación de formato	Tipos de datos aplicables	Ejemplo	Resultado del ejemplo
Número en notación científica	"%e" o "%E"	Flotantes simples o dobles.	<pre>double i=10.5; String d= String.format("Resultado: %E", i); System.out.println (d);</pre>	Resultado: 1.050000E+01
Número decimal	"%f"	Flotantes simples o dobles.	<pre>float i=10.5f; String d= String.format("Resultado: %f", i); System.out.println (d);</pre>	Resultado: 10,50000
Número en notación científica o decimal (lo más corto)	"%g" o "%G"	Flotantes simples o dobles. El número se mostrará como decimal o en notación científica dependiendo de lo que sea mas corto.	<pre>double i=10.5; String d= String.format("Resultado: %g", i); System.out.println (d);</pre>	Resultado: 10.5000

Ahora que ya hemos visto alguna de las conversiones existentes (las más importantes), veamos algunos modificadores que se le pueden aplicar a las conversiones, para ajustar como queremos que sea la salida. Los modificadores se sitúan entre el carácter de escape ("%) y la letra que indica el tipo de conversión (d, f, g, etc.).

Podemos especificar, por ejemplo, el número de caracteres que tendrá como mínimo la salida de una conversión. Si el dato mostrado no llega a ese ancho en caracteres, se llenará con espacios (salvo que se especifique lo contrario):

%[Ancho]Conversión

El hecho de que esté entre corchetes significa que es opcional. Si queremos por ejemplo que la salida genere al menos 5 caracteres (poniendo espacios delante) podríamos ponerlo así:

String.format ("%5d", 10);

Se mostrará el "10" pero también se añadirán 3 espacios delante para llenar. Este tipo de modificador se puede usar con cualquier conversión.

Cuando se trata de conversiones de tipo numéricas con decimales, solo para tipos de datos que admitan decimales, podemos indicar también la precisión, que será el número de decimales mínimos que se mostrarán:

Como puedes ver, tanto el ancho como la precisión van entre corchetes, los corchetes no hay que ponerlos, solo indican que son modificaciones opcionales. Si queremos, por ejemplo, que la salida genere 3 decimales como mínimo, podremos ponerlo así:

```
String.format ("% .3f", 4.2f);
```

Como el número indicado como parámetro solo tiene un decimal, el resultado se completará con ceros por la derecha, generando una cadena como la siguiente: "4,200".

Una cadena de formato puede contener varios especificadores de formato y varios argumentos. Veamos un ejemplo de una cadena con varios especificadores de formato:

```
String np="Lavadora";  
  
int u=10;  
  
float ppu = 302.4f;  
  
float p=u*ppu;  
  
String output=String.format("Producto: %s; Unidades: %d; Precio por unidad: %.2f €; Total: %.2f €", np, u, ppu, p);  
  
System.out.println(output);
```

Cuando el orden de los argumentos es un poco complicado, porque se reutilizan varias veces en la cadena de formato los mismos argumentos, se puede recurrir a los índices de argumento. Se trata de especificar la posición del argumento a utilizar, indicando la posición del argumento (el primer argumento sería el 1 y no el 0) seguido por el símbolo del dólar ("\$"). El índice se ubicaría al comienzo del especificador de formato, después del porcentaje, por ejemplo:

```
int i=10;  
  
int j=20;  
  
String d=String.format("%1$d multiplicado por %2$d (%1$dx%2$d) es %3$d", i,j,i*j);  
  
System.out.println(d);
```

El ejemplo anterior mostraría por pantalla la cadena "10 multiplicado por 20 (10x20) es 200". Los índices de argumento se pueden usar con todas las conversiones, y es compatible con otros modificadores de formato (incluida la precisión).

## Para saber más

Si quieras profundizar en los especificadores de formato puedes acceder a la siguiente página (en inglés), donde encontrarás información adicional acerca de la sintaxis de los especificadores de formato en Java:

[Sintaxis de los especificadores de formato.](#)

## Caso práctico



Ministerio de Educación y FP [\(CC BY-NC\)](#)

En las últimas semanas, **María** y **Juan** han avanzado muchísimo a lo largo de su recorrido por las **estructuras de almacenamiento** más básicas. María sabe que Java dispone de librerías para solventar los problemas de utilizar estructuras de datos estáticas pero es consciente de que debe conocer en profundidad algunos conceptos sobre la **Programación Orientada a Objetos**. Aún recuerda que cuando aprendió a escribir sus propias clases, con sus **atributos** y sus **métodos**, se quedaron muchos conceptos sin terminar de aclarar y que serían estudiados más adelante: utilización de la **herencia**, creación de **interfaces**, **clases abstractas**, **jerarquías de clases**, etc. Estos conceptos son fundamentales para abordar dos temas muy importantes para cualquier desarrollador Java: las colecciones de objetos y las interfaces gráficas de usuario.

Ambos saben que faltan unos cuantos conceptos por asimilar y que sin duda les van a proporcionar más herramientas a la hora de desarrollar sus proyectos. Parece que ha llegado el momento de formalizar algunos de estos conocimientos para poder emplearlos en sus programas.



[Ministerio de Educación y Formación Profesional.](#) (Dominio público)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Relaciones entre clases.

Cuando estudiaste el concepto de clase, ésta fue descrita como una especie de mecanismo de definición (plantillas), en el que se basaría el entorno de ejecución a la hora de construir un objeto: un **mecanismo de definición de objetos**.

Por tanto, a la hora de diseñar un conjunto de clases para modelar el conjunto de información cuyo tratamiento se desea automatizar, es importante establecer apropiadamente las posibles relaciones que puedan existir entre unas clases y otras.

En algunos casos es posible que no exista relación alguna entre unas clases y otras, pero lo más habitual es que sí exista: una clase puede ser una **especialización** de otra, o bien una **generalización**, o una clase contiene en su interior objetos de otra, o una clase utiliza a otra, etc. Es decir, que entre unas clases y otras habrá que definir cuál es su relación (si es que existe alguna).

Se pueden distinguir diversos tipos de relaciones entre clases:

- ✓ **Clientela.** Cuando una clase utiliza objetos de otra clase (por ejemplo al pasárselos como parámetros a través de un método).
- ✓ **Composición.** Cuando alguno de los atributos de una clase es un objeto de otra clase.
- ✓ **Anidamiento.** Cuando se definen clases en el interior de otra clase.
- ✓ **Herencia.** Cuando una clase comparte determinadas características con otra (clase base), añadiéndole alguna funcionalidad específica (especialización).

La relación de **clientela** la llevas utilizando desde que has empezado a programar en Java, pues desde tu clase principal (clase con método `main`) has estado declarando, creando y utilizando objetos de otras clases. Por ejemplo: si utilizas un objeto `String` dentro de la clase principal de tu programa, éste será cliente de la clase `String` (como sucederá con prácticamente cualquier programa que se escriba en Java). Es la relación fundamental y más habitual entre clases (la utilización de unas clases por parte de otras) y, por supuesto, la que más vas a utilizar tú también, de hecho, ya la has estado utilizando y lo seguirás haciendo.

La relación de **composición** es posible que ya la hayas tenido en cuenta si has definido clases que contenían (tenían como atributos) otros objetos en su interior, lo cual es bastante habitual. Por ejemplo, si escribes una clase donde alguno de sus atributos es un objeto de tipo `String`, ya se está produciendo una relación de tipo **composición** (tu clase “tiene” un `String`, es decir, está compuesta por un objeto `String` y por algunos elementos más).



ITE. Félix Vallés Calvo idITE=152018 (CC BY-SA)

La relación de **anidamiento** (o **anidación**) es quizás menos habitual, pues implica declarar unas clases dentro de otras (**clases internas** o **anidadas**). En algunos casos puede resultar útil para tener un nivel más de **encapsulamiento** y **ocultación** de información.

En el caso de la relación de **herencia** también la has visto ya, pues seguro que has utilizado unas clases que derivaban de otras, sobre todo, en el caso de los objetos que forman parte de las **interfaces gráficas**. Lo más probable es que hayas tenido que declarar clases que derivaban de algún componente gráfico (`JFrame`, `JDialog`, etc.).

Podría decirse que tanto la **composición** como la **anidación** son casos particulares de **clientela**, pues en realidad en todos esos casos una clase está haciendo uso de otra (al contener atributos que son objetos de la otra clase, al definir clases dentro de otras clases, al utilizar objetos en el paso de parámetros, al declarar variables locales utilizando otras clases, etc.).

A lo largo de la unidad, irás viendo distintas posibilidades de implementación de clases haciendo uso de todas estas relaciones, centrándonos especialmente en el caso de la **herencia**, que es la que permite establecer las relaciones más complejas.

## Autoevaluación

¿Cuál crees que será la relación entre clases más habitual?

- Clientela.
- Anidación o anidamiento.
- Herencia.
- Entre las clases no existen relaciones. Son entidades aisladas en el sistema y sin relaciones con el exterior.

¡Así es! ¡Enhorabuena! Todas las clases hacen uso unas de otras.



ITE. Félix Vallés Calvo idITE=152093 (CC BY-NC-SA)

Incorrecto. Aunque pueden existir definiciones de unas clases dentro de otras, no es lo más habitual.

¡Has fallado! Aunque la herencia es una de las relaciones más habituales entre clases, existen otro tipo de relaciones aún más habituales.

¡Falso! Entre las clases de un programa existen siempre multitud de relaciones.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 1.1.- Composición.

Cuando en un sistema de información, una determinada entidad A contiene a otra B como una de sus partes, se suele decir que se está produciendo una relación de **composición**. Es decir, el objeto de la clase A contiene a uno o varios objetos de la clase B.

Por ejemplo, si describes una entidad **País** compuesta por una serie de atributos, entre los cuales se encuentra una lista de comunidades autónomas, podrías decir que los objetos de la clase **País** contienen varios objetos de la clase **ComunidadAutonoma**. Por otro lado, los objetos de la clase **ComunidadAutonoma** podrían contener como atributos objetos de la clase **Provincia**, la cual a su vez también podría contener objetos de la clase **Municipio**.



s.volenszki (CC BY-NC)

Como puedes observar, la **composición** puede encadenarse todas las veces que sea necesario hasta llegar a objetos básicos del lenguaje o hasta tipos primitivos que ya no contendrán otros objetos en su interior. Ésta es la forma más habitual de definir clases: mediante otras clases ya definidas anteriormente. Es una manera eficiente y sencilla de gestionar la reutilización de todo el código ya escrito. Si se definen clases que describen entidades distinguibles y con funciones claramente definidas, podrán utilizarse cada vez que haya que representar objetos similares dentro de otras clases.

La **composición** se da cuando una clase contiene algún atributo que es una referencia a un objeto de otra clase.

Una forma sencilla de plantearte si la relación que existe entre dos clases A y B es de **composición** podría ser mediante la expresión idiomática “**tiene un**”: “la clase A tiene uno o varios objetos de la clase B”, o visto de otro modo: “Objetos de la clase B pueden formar parte de la clase A”.

Algunos ejemplos de composición podrían ser:

- ✓ Un **coche** tiene un **motor** y tiene cuatro **ruedas**.
- ✓ Una **persona** tiene un **nombre**, una **fecha de nacimiento**, una **cuenta bancaria** asociada para ingresar la nómina, etc.
- ✓ Un **cocodrilo** bajo investigación científica que tiene un número de **dientes** determinado, una **edad**, unas **coordenadas** de ubicación geográfica (medidas con **GPS**), etc.

Recuperando algunos de los ejemplos de clases que has utilizado en otras unidades:

- ✓ Una clase **Rectangulo** podría contener en su interior dos objetos de la clase **Punto** para almacenar los vértices inferior izquierdo y superior derecho.
- ✓ Una clase **Empleado** podría contener en su interior un objeto de la clase **DNI** para almacenar su DNI/NIF, y otro objeto de la clase **CuentaBancaria** para guardar la cuenta en la que se realizan los ingresos en nómina.

### Ejercicio resuelto

¿Podría decirse que la relación que existe entre la clase **Ave** y la clase **Loro** es una relación de composición?

[Mostrar retroalimentación](#)

No. Aunque claramente existe algún tipo de relación entre ambas, no parece que sea la de composición. No parece que se cumpla la expresión “**tiene un**”: “Un loro tiene un ave”. Se cumpliría más bien una expresión del tipo “**es un**”: “Un loro es un ave”. Algunos objetos que cumplirían la relación de composición podrían ser **Pico** o **Alas**, pues “un loro tiene un pico y dos alas”, del mismo modo que “un ave tiene pico y dos alas”. Este tipo de relación parece más de **herencia** (un loro es un tipo de ave).

## 1.2.- Herencia.

El mecanismo que permite crear clases basándose en otras que ya existen es conocido como **herencia**. Como ya has visto en unidades anteriores, Java implementa la herencia mediante la utilización de la palabra reservada `extends`.



Ryan Somma (CC BY-NC-SA)

El concepto de **herencia** es algo bastante simple y sin embargo muy potente: cuando se desea definir una nueva clase y ya existen clases que, de alguna manera, implementan parte de la funcionalidad que se necesita, es posible crear una nueva **clase derivada** de la que ya tienes. Al hacer esto se posibilita la reutilización de todos los atributos y métodos de la clase que se ha utilizado como **base (clase padre o superclase)**, sin la necesidad de tener que escribirlos de nuevo.

Una **subclase** hereda todos los miembros de su **clase padre** (atributos, métodos y clases internas). Los **constructores** no se heredan, aunque se pueden invocar desde la **subclase**.

Algunos ejemplos de herencia podrían ser:

- ✓ Un **coche** es un **vehículo** (heredará atributos como la **velocidad máxima** o métodos como **parar** y **arrancar**).
- ✓ Un **empleado** es una **persona** (heredará atributos como el **nombre** o la **fecha de nacimiento**).
- ✓ Un **rectángulo** es una **figura geométrica** en el plano (heredará métodos como el cálculo de la **superficie** o de su **perímetro**).
- ✓ Un **cocodrilo** es un **reptil** (heredará atributos como por ejemplo el **número de dientes**).

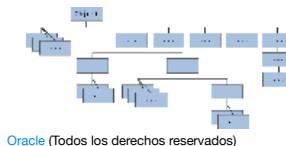
En este caso la expresión idiomática que puedes usar para plantearte si el tipo de relación entre dos clases A y B es de herencia podría ser “**es un**”: “la clase A es un tipo específico de la clase B” (**especialización**), o visto de otro modo: “la clase B es un caso general de la clase A” (**generalización**).

Recuperando algunos ejemplos de clases que ya has utilizado en otras unidades:

- ✓ Una **ventana** en una **aplicación gráfica** puede ser una clase que herede de **JFrame** (componente Swing: `javax.swing.JFrame`), de esta manera esa clase será un marco que dispondrá de todos los métodos y atributos de **JFrame** más aquéllos que tú decidas incorporarle al rellenarlo de componentes gráficos.
- ✓ Una **caja de diálogo** puede ser un tipo de **JDialog** (otro componente Swing: `javax.swing.JDialog`).

En Java, la clase **Object** (dentro del paquete `java.lang`) define e implementa el comportamiento común a todas las clases (incluidas aquellas que tú escribas). Como recordarás, ya se dijo que en Java cualquier clase deriva en última instancia de la clase **Object**.

Todas las clases tienen una **clase padre**, que a su vez también posee una **superclase**, y así sucesivamente hasta llegar a la clase **Object**. De esta manera, se construye lo que habitualmente se conoce como una **jerarquía de clases**, que en el caso de Java tendría a la clase **Object** en la raíz.



### Ejercicio resuelto

Cuando escribes una clase en Java, puedes hacer que herede de una determinada clase padre (mediante el uso de `extends`) o bien no indicar ninguna herencia. En tal caso tu clase no heredará de ninguna otra clase Java. ¿Verdadero o Falso?

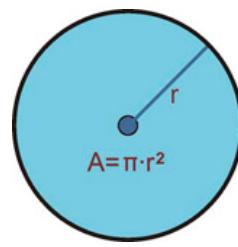
[Mostrar retroalimentación](#)

No es cierto. Aunque no indiques explícitamente ningún tipo de herencia, el compilador asumirá entonces de manera implícita que tu clase hereda de la clase `Object`, que define e implementa el comportamiento común a todas las clases.

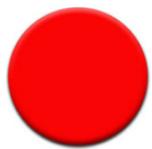
## 1.3.- ¿Herencia o composición?

Cuando esribas tus propias clases, debes intentar tener claro en qué casos utilizar la **composición** y cuándo la **herencia**:

- ✓ **Composición:** cuando una clase está formada por objetos de otras clases. En estos casos se incluyen objetos de esas clases, pero no necesariamente se comparten características con ellos (no se heredan características de esos objetos, sino que directamente se utilizarán sus atributos y sus métodos). Esos objetos incluidos no son más que atributos miembros de la clase que se está definiendo.
- ✓ **Herencia:** cuando una clase cumple todas las características de otra. En estos casos la clase derivada es una **especialización** (o **particularización**, **extensión** o **restricción**) de la clase base. Desde otro punto de vista se diría que la clase base es una **generalización** de las clases derivadas.



ITE\_Antonio.Ortega.Moreno  
idITE=148446 (CC BY-SA)



ITE idITE=168378 (CC BY-NC-SA)

Por ejemplo, imagina que dispones de una clase **Punto** (ya la has utilizado en otras ocasiones) y decides definir una nueva clase llamada **Círculo**. Dado que un punto tiene como atributos sus coordenadas en plano (**x1**, **y1**), decides que es buena idea aprovechar esa información e incorporarla en la clase **Círculo** que estás escribiendo. Para ello utilizas la **herencia**, de manera que al derivar la clase **Círculo** de la clase **Punto**, tendrás disponibles los atributos **x1** e **y1**. Ahora solo faltaría añadirle algunos atributos y métodos más como por ejemplo el **radio** del círculo, el cálculo de su **área** y su **perímetro**, etc.

En principio parece que la idea pueda funcionar pero es posible que más adelante, si continúas construyendo una **jerarquía de clases**, observes que puedes llegar a conclusiones incongruentes al suponer que un círculo es una especialización de un punto (un tipo de punto). ¿Todas aquellas figuras que contengan uno o varios puntos deberían ser tipos de punto? ¿Y si tienes varios puntos? ¿Cómo accedes a ellos? ¿Un rectángulo también tiene sentido que herede de un punto? No parece muy buena idea.



ITE\_Félix.Vallés.Calvo idITE=152400 (CC BY-NC-SA)

Parece que en este caso habría resultado mejor establecer una relación de **composición**. Analízalo detenidamente: ¿cuál de estas dos situaciones te suena mejor?

1. “**Un círculo es un punto** (su centro)”, y por tanto heredará las coordenadas **x1** e **y1** que tiene todo punto. Además tendrá otras características específicas como el **radio** o métodos como el cálculo de la **longitud** de su perímetro o de su **área**.
2. “**Un círculo tiene un punto** (su centro)”, junto con algunos atributos más como por ejemplo el **radio**. También tendrá métodos para el cálculo de su **área** o de la longitud de su **perímetro**.

Parece que en este caso la **composición** refleja con mayor fidelidad la relación que existe entre ambas clases. Normalmente suele ser suficiente con plantearse las preguntas “**¿A es un tipo de B?**” o “**¿A contiene elementos de tipo B?**”.

## 2.- Composición.

### Caso práctico



Ministerio de Educación y FP ([CC BY-NC-SA](#))

**María** es consciente de que las relaciones que pueden existir entre dos clases pueden ser de **clientela**, **composición**, **herencia**, etc. También sabe que una forma muy práctica de distinguir entre la necesidad de una composición de un herencia suele ser mediante las preguntas “**¿Es la clase A un tipo de clase B?**” o “**¿Tiene la clase A elementos de la clase B?**” Normalmente, ese método le suele funcionar a la hora de decidirse por la **composición** o por la **herencia**.

Pero, ¿qué hay que hacer para establecer una relación de composición? ¿Es necesario indicar algún modificador al definir las clases? En tal caso, ¿se indicaría en la clase continente o en la contenida? ¿Afecta de alguna manera al código que hay que escribir? En definitiva, ¿cómo se indica que una clase contiene instancias de otra clase en su interior?

Mientras **María** piensa en voz alta, **Ada** se acerca con una carpeta en la mano y se la entrega: – “Bueno, aquí tienes algunas clases básicas que nos hacen falta para el proyecto de la **Clínica Veterinaria**. A ver qué tal os quedan...”.

## 2.1.- Sintaxis de la composición.

Para indicar que una clase contiene objetos de otra clase no es necesaria **ninguna sintaxis especial**. Cada uno de esos objetos no es más que un atributo y, por tanto, debe ser declarado como tal:

```
class <nombreClase> {  
    [modificadores] <NombreClase1> nombreAtributo1;  
    [modificadores] <NombreClase2> nombreAtributo2;  
    ...  
}
```

En unidades anteriores has trabajado con la clase **Punto**, que definía las coordenadas de un punto en el plano, y con la clase **Rectangulo**, que definía una figura de tipo rectángulo también en el plano a partir de dos de sus **vértices (inferior izquierdo y superior derecho)**. Tal y como hemos formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de **composición: “un rectángulo contiene puntos”**. Por tanto, podrías ahora redefinir los atributos de la clase **Rectangulo** (cuatro **números reales**) como dos objetos de tipo **Punto**:

```
class Rectangulo {  
    private Punto vertice1;  
    private Punto vertice2;  
    ...  
}
```

Ahora los métodos de esta clase deberán tener en cuenta que ya no hay cuatro atributos de tipo **double**, sino dos atributos de tipo **Punto** (cada uno de los cuales contendrá en su interior dos atributos de tipo **double**).

### Autoevaluación

Para declarar un objeto de una clase determinada, como atributo de otra clase, es necesario especificar que existe una relación de composición entre ambas clases mediante el modificador **object**. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

No. El hecho de declarar un atributo como de una clase determinada está ya indicando de manera implícita que existe una relación de composición (un objeto que contiene otros objetos en su interior) sin necesidad de tener que indicar nada más. Además, el modificador **object** no existe. Lo que sí existe es la clase **Object**, de la cual derivan todas las clases en Java.

### Ejercicio resuelto

Intenta escribir los siguientes los métodos de la clase **Rectangulo** teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase **Punto**, en lugar de cuatro elementos de tipo **double**):

1. **Método** calcularSuperficie(), que calcula y devuelve el área de la superficie encerrada por la figura.
2. **Método** calcularPerimetro(), que calcula y devuelve la longitud del perímetro de la figura.

[Mostrar retroalimentación](#)

En ambos casos la **interfaz** no se ve modificada en absoluto (desde fuera su funcionamiento es el mismo), pero internamente deberás tener en cuenta que ya no existen los atributos x1, y1, x2, y2, de tipo double, sino los atributos vertice1 y vertice2 de tipo Punto.

```
public double calcularSuperficie () {  
    double area, base, altura; // Variables locales  
  
    base= vertice2.obtenerX () - vertice1.obtenerX (); // Antes era x2 - x1  
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1  
    area= base * altura;  
  
    return area;  
}  
  
public double CalcularPerimetro () {  
    double perimetro, base, altura; // Variables locales  
  
    base= vertice2.obtenerX () - vertice1.obtenerX (); // Antes era x2 - x1  
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1  
    perimetro= 2*base + 2*altura;  
  
    return perimetro;  
}
```

En la siguiente presentación puedes observar detalladamente el proceso completo de elaboración de la clase Rectangulo haciendo uso de la clase Punto:

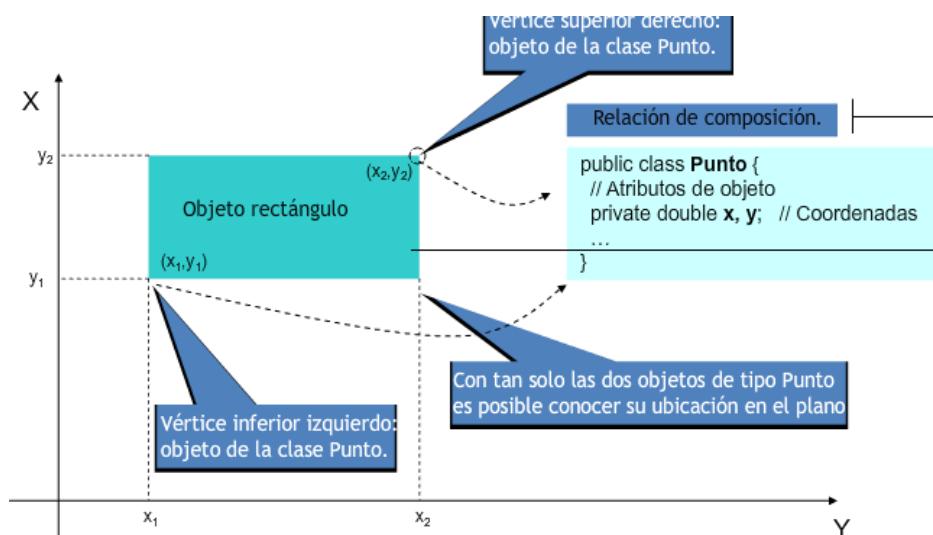


## Clase Rectangulo

## Objetos de tipo Rectángulo compuesto por objetos de tipo Punto

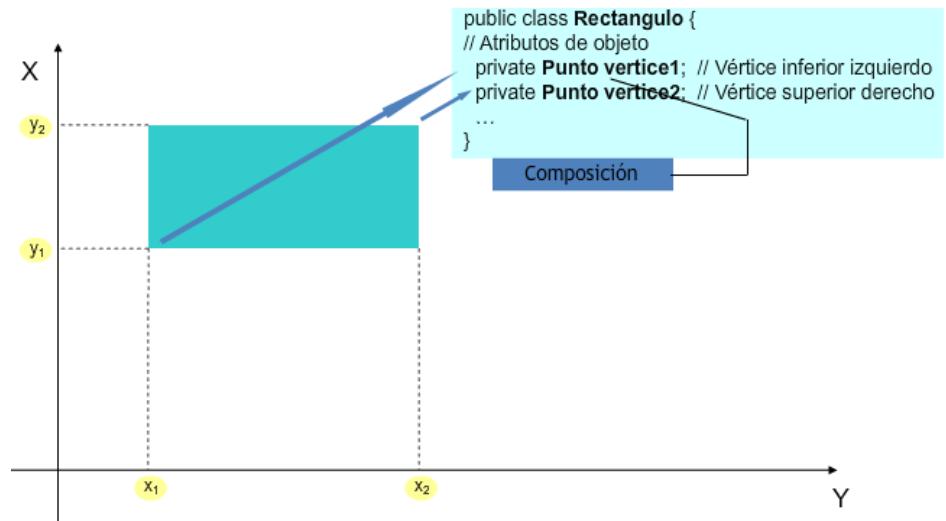
Ministerio de Educación y FP ([CC BY-NC](#))

## Clase Rectangulo II



Ministerio de Educación y FP ([CC BY-NC](#))

## Clase Rectangulo III



Ministerio de Educación y FP ([CC BY-NC](#))

## Clase Rectangulo IV

### Clase rectángulo

Ministerio de Educación y FP ([CC BY-NC](#))

## Clase Rectangulo V

X

A partir de ahora los rectángulos serán representados mediante objetos de esta clase (contenedores de dos objetos Punto).

```
public class Rectangulo {  
    // Atributos de objeto  
    private Punto vertice1; // Vértice inferior izquierdo  
    private Punto vertice2; // Vértice superior derecho  
    ...  
}
```

Y

Ministerio de Educación y FP ([CC BY-NC](#))

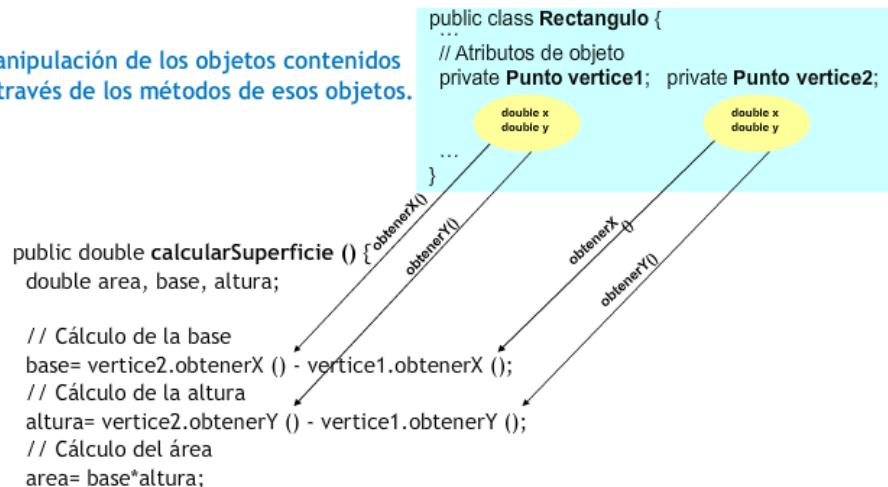
## Clase Rectangulo VI

### Método calcularSuperficie

Ministerio de Educación y FP ([CC BY-NC](#))

## Clase Rectangulo VII

Manipulación de los objetos contenidos a través de los métodos de esos objetos.



Ministerio de Educación y FP ([CC BY-NC](#))

## Clase Rectangulo

Todas las imágenes utilizadas son propiedad del Ministerio de Educación y FP bajo licencia CC BY-NC

[Resumen textual alternativo](#)

## 2.2.- Uso de la composición (I). Preservación de la ocultación.

Como ya has observado, la relación de **composición** no tiene más misterio a la hora de implementarse que simplemente declarar **atributos** de las clases que necesites dentro de la clase que estés diseñando.

Ahora bien, cuando escribes clases que contienen objetos de otras clases (lo cual será lo más habitual) deberás tener un poco de precaución con aquellos métodos que devuelvan información acerca de los **atributos** de la clase (métodos “**obtenedores**” o de tipo **get**).

Como ya viste en la unidad dedicada a la creación de clases, lo normal suele ser declarar los **atributos** como **privados** (o **protegidos**, como veremos un poco más adelante) para ocultarlos a los posibles **clientes** de la clase (otros objetos que en el futuro harán uso de la clase). Para que otros objetos puedan acceder a la información contenida en los **atributos**, o al menos a una parte de ella, deberán hacerlo a través de **métodos que sirvan de interfaz**, de manera que sólo se podrá tener acceso a aquella información que el creador de la clase haya considerado oportuna. Del mismo modo, los **atributos** solamente serán modificados desde los métodos de la clase, que decidirán cómo y bajo qué circunstancias deben realizarse esas modificaciones. Con esa metodología de acceso se tenía perfectamente separada la parte de manipulación interna de los atributos de la **interfaz** con el exterior.

Hasta ahora los métodos de tipo **get** devolvían **tipos primitivos**, es decir, copias del contenido (a veces con algún tipo de modificación o de formato) que había almacenado en los **atributos**, pero los atributos seguían “a salvo” como elementos privados de la clase. Pero, a partir de este momento, al tener objetos dentro de las clases y no sólo tipos primitivos, es posible que en un determinado momento interese devolver un **objeto completo**.

Ahora bien, cuando vayas a devolver un objeto habrás de obrar con mucha precaución. Si en un método de la clase devuelves directamente un objeto que es un atributo, estarás ofreciendo directamente una **referencia** a un objeto atributo que probablemente has definido como privado. ¡De esta forma estás **viviendo a hacer público un atributo que inicialmente era privado!**!

Para evitar ese tipo de situaciones (ofrecer al exterior referencias a objetos privados) puedes optar por diversas alternativas, procurando siempre evitar la devolución directa de un atributo que sea un objeto:

- ✓ Una opción podría ser devolver siempre tipos primitivos.
- ✓ Dado que esto no siempre es posible, o como mínimo poco práctico, otra posibilidad es crear un nuevo objeto que sea una copia del atributo que quieras devolver y utilizar ese objeto como valor de retorno. Es decir, **crear una copia del objeto** especialmente para devolverlo. De esta manera, el código cliente de ese método podrá manipular a su antojo ese nuevo objeto, pues no será una referencia al atributo original, sino un nuevo objeto con el mismo contenido.

Por último, debes tener en cuenta que es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo muy habitual en el caso de atributos estáticos). En tales casos, no habrá problema en devolver directamente el atributo para que el código llamante (cliente) haga el uso que estime oportuno de él.

Debes evitar por todos los medios la devolución de un atributo que sea un objeto (estarías dando directamente una referencia al atributo, visible y manipulable desde fuera), salvo que se trate de un caso en el que deba ser así.

Para entender estas situaciones un poco mejor, podemos volver al objeto **Rectángulo** y observar sus nuevos métodos de tipo **get**.

### Ejercicio resuelto

Dada la clase **Rectángulo**, escribe sus nuevos métodos **obtenerVertice1** y **obtenerVertice2** para que devuelvan los vértices inferior izquierdo y superior derecho del rectángulo (objetos de tipo **Punto**), teniendo en cuenta su nueva estructura de atributos (dos objetos de la clase **Punto**, en lugar de cuatro elementos de tipo **double**):

[Mostrar retroalimentación](#)

Los métodos de obtención de vértices devolverán objetos de la clase **Punto**:

```

public Punto obtenerVertice1 () {
    return vertice1;
}

public Punto obtenerVertice2 () {
    return vertice2;
}

```

Esto funcionaría perfectamente, pero deberías tener cuidado con este tipo de métodos que devuelven directamente una referencia a un objeto atributo que probablemente has definido como privado. Estás de alguna manera haciendo público un atributo que fue declarado como privado.

Para evitar que esto suceda bastaría con crear un nuevo objeto que fuera una copia del atributo que se desea devolver (en este caso un objeto de la clase `Punto`).

Aquí tienes algunas posibilidades:

```

public Punto obtenerVertice1 () // Creación de un nuevo punto extrayendo sus atributos
{
    double x, y;

    Punto p;

    x= this.vertice1.obtenerX();
    y= this.vertice1.obtenerY();

    p= new Punto (x,y);

    return p;
}

public Punto obtenerVertice1 () // Utilizando el constructor copia de Punto (si es que está definido)
{
    Punto p;

    p= new Punto (this.vertice1); // Uso del constructor copia

    return p;
}

```

De esta manera, se devuelve un punto totalmente nuevo que podrá ser manipulado sin ningún temor por parte del código cliente de la clase pues es una copia para él.

Para el método `obtenerVertice2` sería exactamente igual.

## 2.3.- Uso de la composición (II). Llamadas a constructores.

Otro factor que debes considerar, a la hora de escribir clases que contengan como atributos objetos de otras clases, es su comportamiento a la hora de instanciarse. Durante el proceso de creación de un objeto (**constructor**) de la clase contenida habrá que tener en cuenta también la creación (llamadas a **constructores**) de aquellos objetos que son contenidos.

El constructor de la clase contenida debe invocar a los constructores de las clases de los objetos contenidos.

En este caso hay que tener cuidado con las referencias a **objetos que se pasan como parámetros** para llenar el contenido de los atributos. Es conveniente hacer una copia de esos objetos y utilizar esas copias para los atributos pues si se utiliza la referencia que se ha pasado como parámetro, el código cliente de la clase podría tener acceso a ella sin necesidad de pasar por la interfaz de la clase (volveríamos a dejar abierta una **puerta pública** a algo que quizás sea privado).

Además, si el **objeto parámetro** que se pasó al **constructor** formaba parte de otro objeto, esto podría ocasionar un desagradable efecto colateral si esos objetos son modificados en el futuro desde el código cliente de la clase, ya que no sabes de dónde provienen esos objetos, si fueron creados especialmente para ser usados por el nuevo objeto creado o si pertenecen a otro objeto que podría modificarlos más tarde. Es decir, correrías el riesgo de estar “compartiendo” esos objetos con otras partes del código, sin ningún tipo de control de acceso y con las nefastas consecuencias que eso podría tener: cualquier cambio de ese objeto afectaría a partes del programa supuestamente independientes, que entienden ese objeto como suyo.



ITE\_Luana Fischer Ferreira,  
IdITE= 149925 (CC BY-SA)

En el fondo los objetos no son más que variables de tipo referencia a la zona de memoria en la que se encuentra toda la información del objeto en sí mismo. Esto es, puedes tener un único objeto y múltiples referencias a él. Pero sólo se trata de un objeto, y cualquier modificación desde una de sus referencias afectaría a todas las demás, pues estamos hablando del mismo objeto.

Recuerda también que sólo se crean objetos cuando se llama a un constructor (uso de `new`). Si realizas asignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino simplemente de las referencias, y por tanto se tratará siempre del mismo objeto.

Se trata de un efecto similar al que sucedía en los métodos de tipo **get**, pero en este caso en sentido contrario (en lugar de que nuestra clase “regale” al exterior uno de sus atributos objeto mediante una referencia, en esta ocasión se “adueña” de un parámetro objeto que probablemente pertenezca a otro objeto y que es posible que el futuro haga uso de él).

Para entender mejor estos posibles efectos podemos continuar con el ejemplo de la clase `Rectangulo` que contiene en su interior dos objetos de la clase `Punto`. En los constructores del rectángulo habrá que incluir todo lo necesario para crear dos instancias de la clase `Punto` evitando las referencias a parámetros (haciendo copias).

### Autoevaluación

Si se declaran dos variables `a` y `b` de la clase `X`, ambas son instanciadas mediante un constructor, y posteriormente se realiza la asignación `a=b`, el contenido de `b` será una copia del contenido de `a`, perdiéndose los valores iniciales de `b`. ¿Verdadero o Falso?

Verdadero  Falso

#### Falso

No es así. La variable `b` se habrá convertido en una referencia al mismo objeto al que apunta la variable `a`, pues no se ha hecho una copia de `a`. Ahora se tendrán dos variables referencia (`a` y `b`) a un mismo objeto (el que fue inicialmente instanciado y asignado a la variable `b`). Por otro lado, el objeto al que apuntaba la variable `b` se habrá perdido y el recolector de basura se encargará de destruirlo (salvo que existieran otras referencias a él).

## Ejercicio resuelto

Intenta escribir los constructores de la clase `Rectangulo` teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`):

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, `x1`, `y1`, `x2`, `y2`, que cree un rectángulo con los vértices (`x1`, `y1`) y (`x2`, `y2`).
3. Un constructor con dos parámetros, `punto1`, `punto2`, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, `base` y `altura`, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.

[Mostrar retroalimentación](#)

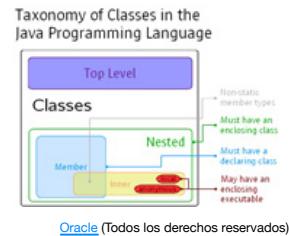
En el siguiente documento puedes observar el proceso completo de elaboración de todos los constructores de la clase:

[Proceso de elaboración de los constructores de la clase `Rectangulo`.](#)

## 2.4.- Clases anidadas o internas.

En algunos lenguajes, es posible definir una clase dentro de otra clase (**clases internas**):

```
class claseContenedora {  
    // Cuerpo de la clase  
    ...  
    class claseInterna {  
        // Cuerpo de la clase interna  
        ...  
    }  
}
```



Se pueden distinguir varios tipos de **clases internas**:

- ✓ **Clases internas estáticas** (o **clases anidadas**), declaradas con el modificador `static`.
- ✓ **Clases internas miembro**, conocidas habitualmente como **clases internas**. Declaradas al máximo nivel de la clase contenedora y no estáticas.
- ✓ **Clases internas locales**, que se declaran en el interior de un bloque de código (normalmente dentro de un método).
- ✓ **Clases anónimas**, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la **gestión de eventos** en los **interfaces gráficos**.

Aquí tienes algunos ejemplos:

```
class claseContenedora {  
    ...  
    static class claseAnidadaEstatica {  
        ...  
    }  
    class claseInterna {  
        ...  
    }  
}
```

Las **clases anidadas**, como miembros de una clase que son (miembros de `claseExterna`), pueden ser declaradas con los modificadores `public`, `protected`, `private` o **de paquete**, como el resto de miembros.

Las **clases internas** (no estáticas) tienen acceso a otros miembros de la clase dentro de la que está definida aunque sean privados (se trata en cierto modo de un miembro más de la clase), mientras que las anidadas (estáticas) no.



Las **clases internas** se utilizan en algunos casos para:

- ✓ **Agrupar** clases que sólo tiene sentido que existan en el entorno de la clase en la que han sido definidas, de manera que se oculta su existencia al resto del código.
- ✓ Incrementar el nivel de **encapsulación** y **ocultamiento**.
- ✓ Proporcionar un **código fuente más legible y fácil de mantener** (el código de las **clases internas** y **anidadas** está más cerca de donde es usado).

En Java es posible definir **clases internas** y **anidadas**, permitiendo todas esas posibilidades. Aunque para lo ejemplos con los que vas a trabajar no las vas a necesitar por ahora.

## Para saber más

También puedes consultar el siguiente artículo sobre clases de alto nivel, clases miembro, clases internas y clases anidadas (en inglés):

[Nested Inner, Member, and Top-Level Classes.](#)

## 3.- Herencia.

### Caso práctico

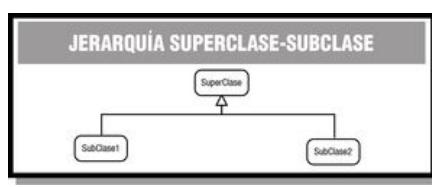
**María** ha estado desarrollando junto con **Juan** algunas clases para el proyecto de la **Clínica Veterinaria**. Hasta el momento todo ha ido bien. Han tenido que crear clases que contenían en su interior instancias de otras clases (atributos que eran objetos). Han tenido cuidado con los **constructores** y las **referencias** a los atributos internos y parece que, por ahora, todo funciona perfectamente. Pero ahora necesitan aprovechar algunas de las características que tienen algunas de las clases que ya han escrito, y no quieren tener que volver a escribir todos esos métodos en las nuevas clases. **María** sabe que es una ocasión perfecta para utilizar el concepto de herencia:



Ministerio de Educación y FP (CC BY-NC)

– “Si la clase A tiene características en común (atributos y métodos) con la clase B aportando algunas características nuevas, puede decirse que la **clase A es una especialización de la clase B**, ¿no es así?”. – Le pregunta **María** a **Juan**.– “Así es. Es un caso claro de **herencia**. La clase A hereda de la clase B”. – Contesta **Juan**.– “De acuerdo. Pues vamos manos a la obra. ¿Cómo indicábamos que una clase heredaba de otra? Creo recordar que se usaba la palabra reservada `extends`. ¿Había que hacer algo más?” – Dice **María** con entusiasmo.– “Parece que ha llegado el momento de repasar la sintaxis de la **herencia** en Java”.

Como ya has estudiado, la **herencia** es el mecanismo que permite definir una nueva clase a partir de otra, pudiendo añadir nuevas características, sin tener que volver a escribir todo el código de la clase base.



Ministerio de Educación y FP (CC BY-NC)

La clase de la que se hereda suele ser llamada **clase base, clase padre o superclase**. A la clase que hereda se le suele llamar **clase hija, clase derivada o subclase**.

Una clase derivada puede ser a su vez **clase padre** de otra que herede de ella y así sucesivamente dando lugar a una **jerarquía de clases**, excepto aquellas que estén en la parte de arriba de la jerarquía (sólo serán **clases padre**) o en la parte de abajo (sólo serán **clases hijas**).

Una **clase hija** no tiene acceso a los miembros **privados** de su **clase padre**, tan solo a los **públicos** (como cualquier parte del código tendría) y los **protegidos** (a los que sólo tienen acceso las **clases derivadas** y las del mismo **paquete**). Aquellos miembros que sean privados en la clase base también habrán sido heredados, pero el acceso a ellos estará restringido al propio funcionamiento de la **superclase** y sólo se podrá acceder a ellos si la **superclase** ha dejado algún medio indirecto para hacerlo (por ejemplo a través de algún método).

Todos los miembros de la **superclase**, tanto atributos como métodos, son heredados por la subclase. Algunos de estos miembros heredados podrán ser **redefinidos** o **sobrescritos (overriden)** y también podrán añadirse nuevos miembros. De alguna manera podría decirse que estás “ampliando” la **clase base** con características adicionales o modificando algunas de ellas (proceso de **especialización**).

Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir su código, es decir, de forma automática la clase derivada hereda sus propiedades y sus métodos.

### Autoevaluación

Una clase derivada hereda todos los miembros de su clase base, pudiendo acceder a cualquiera de

ellos en cualquier momento. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

No es del todo cierto. Se heredan todos los miembros, pero la clase derivada no tendrá un acceso directo a aquellos miembros de la clase base que sean privados.

### 3.1.- Sintaxis de la herencia.

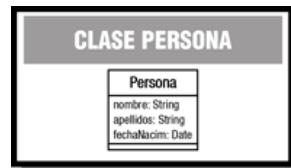
En Java la **herencia** se indica mediante la palabra reservada `extends`:

```
[modificador] class ClasePadre {  
    // Cuerpo de la clase  
    ...  
}  
  
[modificador] class ClaseHija extends ClasePadre {  
    // Cuerpo de la clase  
    ...  
}
```

Imagina que tienes una clase **Persona** que contiene atributos como **nombre**, **apellidos** y **fecha de nacimiento**:

```
public class Persona {  
    String nombre;  
    String apellidos;  
    GregorianCalendar fechaNacim;  
    ...  
}
```

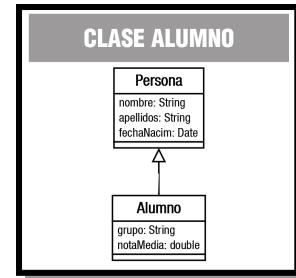
Es posible que, más adelante, necesites una clase **Alumno** que compartirá esos atributos (dado que todo alumno es una persona, pero con algunas características específicas que lo **especializan**). En tal caso tendrías la posibilidad de crear una clase **Alumno** que repitiera todos esos atributos o bien **heredar** de la clase **Persona**:



Ministerio de Educación y FP ([CC BY-NC](#))

```
public class Alumno extends Persona {  
    String grupo;  
    double notaMedia;  
    ...  
}
```

A partir de ahora, un objeto de la clase **Alumno** contendrá los atributos **grupo** y **notaMedia** (propios de la clase **Alumno**), pero también **nombre**, **apellidos** y **fechaNacim** (propios de su **clase base** **Persona** y que por tanto ha heredado).



Ministerio de Educación y FP [\(CC BY-NC\)](#)

## Autoevaluación

En Java la herencia se indica mediante la palabra reservada `inherits`. ¿Verdadero o Falso?

- Verdadero  Falso

Falso

La herencia se indica mediante la palabra reservada `extends`.

## Ejercicio resuelto

Imagina que también necesitas una clase **Profesor**, que contará con atributos como nombre, apellidos, fecha de nacimiento, salario y especialidad. ¿Cómo crearías esa nueva clase y qué atributos le añadirías?

[Mostrar retroalimentación](#)

Está claro que un **Profesor** es otra especialización de **Persona**, al igual que lo era **Alumno**, así que podrías crear otra clase derivada de **Persona** y así aprovechar los atributos genéricos (**nombre**, **apellidos**, **fecha de nacimiento**) que posee todo objeto de tipo **Persona**. Tan solo faltaría añadirle sus atributos específicos (**salario** y **especialidad**):

```

public class Profesor extends Persona {

    String especialidad;

    double salario;

    ...

}

```

## 3.2.- Acceso a miembros heredados.

Como ya has visto anteriormente, no es posible acceder a miembros **privados** de una superclase. Para poder acceder a ellos podrías pensar en hacerlos **públicos**, pero entonces estarías dando la opción de acceder a ellos a cualquier objeto externo y es probable que tampoco sea eso lo deseable. Para ello se inventó el modificador **protected** (**protegido**) que permite el **acceso desde clases heredadas**, pero no desde fuera de las clases (estrictamente hablando, desde fuera del **paquete**), que serían como miembros **privados**.

En la unidad dedicada a la utilización de clases ya estudiaste los posibles modificadores de acceso que podía tener un miembro: **sin modificador** (acceso de paquete), **público**, **privado** o **protegido**. Aquí tienes de nuevo el resumen:

Cuadro de niveles accesibilidad a los atributos de una clase

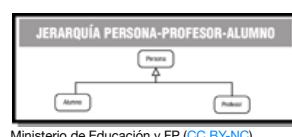
	Misma clase	Subclase	Mismo paquete	Otro paquete
<b>Sin modificador (paquete)</b>	X		X	X
<b>public</b>	X	X	X	X
<b>private</b>	X			
<b>protected</b>	X	X	X	

Si en el ejemplo anterior de la clase **Persona** se hubieran definido sus atributos como **private**:

```
public class Persona {  
    private String nombre;  
    private String apellidos;  
    ...  
}
```

Al definir la clase **Alumno** como heredera de **Persona**, no habrías tenido acceso a esos atributos, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como **protected** o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si es que se considera oportuno):

```
public class Persona {  
    protected String nombre;  
    protected String apellidos;  
    ...  
}
```



Ministerio de Educación y FP (CC BY-NC)

Sólo en aquellos casos en los que se desea explícitamente que un miembro de una clase no pueda ser accesible desde una clase derivada debería utilizarse el modificador **private**. En el resto de casos es recomendable utilizar **protected**, o bien no indicar modificador (acceso a nivel de **paquete**).

### Ejercicio resuelto

**Describe las clases Alumno y Profesor utilizando el modificador `protected` para sus atributos del mismo modo que se ha hecho para su superclase Persona**

[Mostrar retroalimentación](#)

**1. Clase Alumno.**

Se trata simplemente de añadir el modificador de acceso `protected` a los nuevos atributos que añade la clase.

```
public class Alumno extends Persona {  
    protected String grupo;  
    protected double notaMedia;  
    ...  
}
```

**2. Clase Profesor.**

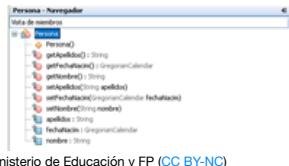
Exactamente igual que en la clase Alumno.

```
public class Profesor extends Persona {  
    protected String especialidad;  
    protected double salario;  
    ...  
}
```

### 3.3.- Utilización de miembros heredados (I). Atributos.

Los **atributos heredados** por una clase son, a efectos prácticos, iguales que aquellos que sean definidos específicamente en la nueva **clase derivada**.

En el ejemplo anterior la clase `Persona` disponía de tres atributos y la clase `Alumno`, que heredaba de ella, añadía dos atributos más. Desde un punto de vista funcional podrías considerar que la clase `Alumno` tiene cinco atributos: tres por ser `Persona` (**nombre, apellidos, fecha de nacimiento**) y otros dos más por ser `Alumno` (**grupo** y **nota media**).



Ministerio de Educación y FP ([CC BY-NC](#))

#### Ejercicio resuelto

Dadas las clases `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos `get` y `set` en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos).

[Mostrar retroalimentación](#)

En el siguiente enlace puedes observar el proceso completo de elaboración de todos los métodos:

[Métodos `get` y `set` para los atributos de las clases `Alumno` y `Profesor`, que heredan de `Persona`.](#)

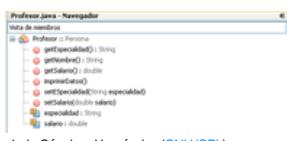
### 3.3.1- Utilización de miembros heredados (II). Métodos.

Del mismo modo que se heredan los **atributos**, también se heredan los **métodos**, convirtiéndose a partir de ese momento en otros **métodos** más de la **clase derivada**, junto a los que hayan sido definidos específicamente.

En el ejemplo de la clase **Persona**, si dispusiéramos de métodos **get** y **set** para cada uno de sus tres atributos (**nombre**, **apellidos**, **fechaNacim**), tendrías seis métodos que podrían ser heredados por sus **clases derivadas**. Podrías decir entonces que la clase **Alumno**, derivada de **Persona**, tiene diez métodos:

- ✓ Seis por ser **Persona** (**getNombre**, **getApellidos**, **getFechaNacim**, **setNombre**, **setApellidos**, **setFechaNacim**).
- ✓ Oros cuatro más por ser **Alumno** (**getGrupo**, **setGrupo**, **getNotaMedia**, **setNotaMedia**).

Sin embargo, sólo tendrías que definir esos cuatro últimos (los **específicos**) pues los **genéricos** ya los has heredado de la **superclase**.



Diosdado Sánchez Hernández ([GNU/GPL](#))

#### Autoevaluación

En Java los métodos heredados de una superclase deben volver a ser definidos en las subclases.  
¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

Los métodos heredados de la superclase están disponibles en la subclase sin tener que hacer nada especial, aunque es cierto, como verás más adelante, que pueden ser redefinidos o ampliados (pero no es obligatorio).

#### Ejercicio resuelto

Dadas las clases **Persona**, **Alumno** y **Profesor** que has utilizado anteriormente, implementa métodos **get** y **set** en la clase **Persona** para trabajar con sus tres atributos y en las clases **Alumno** y **Profesor** para manipular sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para **Persona** van a ser heredados en **Alumno** y en **Profesor**.

[Mostrar retroalimentación](#)

En el siguiente enlace puedes observar el proceso completo de elaboración de todos los métodos:

[Métodos \*\*get\*\* y \*\*set\*\* para los atributos de las clases \*\*Alumno\*\* y \*\*Profesor\*\*, que heredan de \*\*Persona\*\*.](#)

## 3.4.- Redefinición de métodos heredados.

Una clase puede **redefinir** algunos de los métodos que ha heredado de su **clase base**. En tal caso, el nuevo método (**especializado**) sustituye al **heredado**. Este procedimiento también es conocido como **sobrescritura de métodos**.

En cualquier caso, aunque un método sea **sobrescrito** o **redefinido**, aún es posible acceder a él a través de la referencia `super`, aunque sólo se podrá acceder a métodos de la **clase padre** y no a métodos de clases superiores en la **jerarquía de herencia**.

Los **métodos redefinidos** pueden **ampliar su accesibilidad** con respecto a la que ofrece el método original de la **superclase**, pero **nunca restringirla**. Por ejemplo, si un método es declarado como `protected` o **de paquete** en la clase base, podría ser redefinido como `public` en una clase derivada.



Ministerio de Educación y FP.  
[\(CC BY-NC\)](https://creativecommons.org/licenses/by-nc/3.0/)

Los **métodos estáticos** o de clase no pueden ser sobrescritos. Los originales de la clase base permanecen inalterables a través de toda la **jerarquía de herencia**.

En el ejemplo de la clase `Alumno`, podrían redefinirse algunos de los métodos heredados. Por ejemplo, imagina que el método `getApellidos` devuelva la cadena “Alumno:” junto con los apellidos del alumno. En tal caso habría que escribir ese método para realizar esa modificación:

```
public String getApellidos () {  
    return "Alumno: " + apellidos;  
}
```

Cuando sobrescribas un método heredado en Java puedes incluir la **anotación @Override**. Esto indicará al compilador que tu intención es **sobrescribir el método de la clase padre**. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobrescribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar `@Override`, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobrescrito un **método heredado** y al confundirte en una letra estás realmente creando un nuevo método diferente). En el caso del ejemplo anterior quedaría:

```
@Override  
public String getApellidos () {}
```

### Autoevaluación

Dado que el método `finalize()` de la clase `Object` es `protected`, el método `finalize()` de cualquier clase que tú escribas podrá ser `public`, `private` o `protected`. ¿Verdadero o Falso?

Verdadero  Falso

**Falso**

Sabemos que en Java toda clase acaba siempre derivando de la clase `Object` y también sabemos que cuando se redefine un método puede ampliarse su accesibilidad, pero nunca restringirse. De este modo, si el método original (el de la clase en la que se definió) es `protected`, podrá ampliarse su accesibilidad a `public` o dejarse como `protected`, pero nunca restringirse más.

# Ejercicio resuelto

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, redefine el método `getNombre` para que devuelva la cadena “`Alumno:`”, junto con el nombre del alumno, si se trata de un objeto de la clase `Alumno` o bien “`Profesor`”, junto con el nombre del profesor, si se trata de un objeto de la clase `Profesor`.

[Mostrar retroalimentación](#)

## 1. Clase Alumno.

Al heredar de la clase `Persona` tan solo es necesario escribir métodos para los nuevos atributos (**métodos especializados** de acceso a los **atributos especializados**), pues los **métodos genéricos** (de acceso a los **atributos genéricos**) ya forman parte de la clase al haberlos heredado. Esos son los métodos que se implementaron en el ejercicio anterior (`getGrupo`, `setGrupo`, etc.).

Ahora bien, hay que escribir otro método más, pues tienes que redefinir el método `getNombre` para que tenga un comportamiento un poco diferente al `getNombre` que se hereda de la clase base `Persona`:

```
// Método getNombre  
  
@Override  
  
public String getNombre (){  
  
    return "Alumno: " + this.nombre;  
  
}
```

En este caso podría decirse que se “renuncia” al método heredado para redefinirlo con un comportamiento más especializado y acorde con la clase derivada.

## 2. Clase Profesor.

Seguimos exactamente el mismo procedimiento que con la clase `Alumno` (redefinición del método `getNombre`).

```
// Método getNombre  
  
@Override  
  
public String getNombre (){  
  
    return "Profesor: " + this.nombre;  
  
}
```

## 3.5.- Ampliación de métodos heredados.

Hasta ahora, has visto que para **redefinir** o **sustituir** un **método** de una **superclase** es suficiente con crear otro método en la **subclase** que tenga el mismo nombre que el método que se desea **sobrescribir**. Pero, en otras ocasiones, puede que lo que necesites no sea sustituir completamente el comportamiento del método de la superclase, sino simplemente **ampliarlo**.

Para poder hacer esto necesitas poder **preservar el comportamiento antiguo** (el de la **superclase**) y **añadir el nuevo** (el de la **subclase**). Para ello, puedes invocar desde el método “**ampliador**” de la **clase derivada** al método “**ampliado**” de la clase superior (teniendo ambos métodos el mismo nombre). ¿Cómo se puede conseguir eso? Puedes hacerlo mediante el uso de la referencia `super`.

La palabra reservada `super` es una referencia a la **clase padre** de la clase en la que te encuentres en cada momento (es algo similar a `this`, que representaba una referencia a la **clase actual**). De esta manera, podrías invocar a cualquier método de tu **superclase** (si es que se tiene acceso a él).

Por ejemplo, imagina que la clase `Persona` dispone de un método que permite mostrar el contenido de algunos datos personales de los objetos de este tipo (**nombre**, **apellidos**, etc.). Por otro lado, la clase `Alumno` también necesita un método similar, pero que muestre también su información especializada (**grupo**, **nota media**, etc.). ¿Cómo podrías aprovechar el método de la **superclase** para no tener que volver a escribir su contenido en la subclase?

Podría hacerse de una manera tan sencilla como la siguiente:

```
public void mostrar () {  
    super.mostrar (); // Llamada al método “mostrar” de la superclase  
  
    // A continuación mostramos la información “especializada” de esta subclase  
  
    System.out.printf ("Grupo: %s\n", this.grupo);  
  
    System.out.printf ("Nota media: %.2f\n", this.notaMedia);  
}
```

Este tipo de **ampliaciones de métodos** resultan especialmente útiles por ejemplo en el caso de los **constructores**, donde se podría ir llamando a los **constructores** de cada **superclase** encadenadamente hasta el **constructor** de la clase en la **cúspide de la jerarquía** (el **constructor** de la clase `Object`).



ITE (CC BY-SA)

### Ejercicio resuelto

Dadas las clases `Persona`, `Alumno` y `Profesor`, define un método **mostrar** para la clase `Persona`, que muestre el contenido de los atributos (datos personales) de un objeto de la clase `Persona`. A continuación, define sendos métodos **mostrar** especializados para las clases `Alumno` y `Profesor` que “amplíen” la funcionalidad del método **mostrar** original de la clase `Persona`.

[Mostrar retroalimentación](#)

1. Método **mostrar** de la clase `Persona`.

```
public void mostrar () {  
  
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");  
  
    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());  
  
    System.out.printf ("Nombre: %s\n", this.nombre);  
  
    System.out.printf ("Apellidos: %s\n", this.apellidos);
```

```
        System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);  
    }  
  
}
```

2. Método **mostrar** de la clase **Profesor**.

Llamamos al método **mostrar** de su **clase padre** (**Persona**) y luego añadimos la **funcionalidad específica** para la **subclase Profesor**:

```
public void mostrar () {  
  
    super.mostrar (); // Llamada al método “mostrar” de la superclase  
  
    // A continuación mostramos la información “especializada” de esta subclase  
  
    System.out.printf ("Especialidad: %s\n", this.especialidad);  
  
    System.out.printf ("Salario: %7.2f euros\n", this.salario);  
  
}
```

3. Método **mostrar** de la clase **Alumno**.

Llamamos al método **mostrar** de su **clase padre** (**Persona**) y luego añadimos la **funcionalidad específica** para la **subclase Alumno**:

```
public void mostrar () {  
  
    super.mostrar ();  
  
    // A continuación mostramos la información “especializada” de esta subclase  
  
    System.out.printf ("Grupo: %s\n", this.grupo);  
  
    System.out.printf ("Nota media: %5.2f\n", this.notaMedia);  
  
}
```

## 3.6.- Constructores y herencia.

Recuerda que cuando estudiaste los **constructores** viste que un **constructor** de una clase puede llamar a otro **constructor** de la misma clase, previamente definido, a través de la referencia `this`. En estos casos, la utilización de `this` sólo podía hacerse en la primera línea de código del **constructor**.

Como ya has visto, un **constructor** de una **clase derivada** puede hacer algo parecido para llamar al **constructor** de su **clase base** mediante el uso de la palabra `super`. De esta manera, el **constructor** de una **clase derivada** puede llamar primero al **constructor** de su **superclase** para que inicialice los **atributos heredados** y posteriormente se inicializarán los **atributos específicos** de la clase: los no heredados. Nuevamente, esta llamada también **debe ser la primera sentencia de un constructor** (con la única excepción de que exista una llamada a otro constructor de la clase mediante `this`).



Ethorson (CC BY-NC-SA)

Si no se incluye una llamada a `super()` dentro del **constructor**, el compilador incluye automáticamente una llamada al constructor por defecto de **clase base** (llamada a `super()`). Esto da lugar a una **Llamada en cadena de constructores de superclase** hasta llegar a la clase más alta de la jerarquía (que en Java es la clase `Object`).

En el caso del **constructor por defecto** (el que crea el compilador si el programador no ha escrito ninguno), el compilador añade lo primero de todo, antes de la inicialización de los atributos a sus valores por defecto, una llamada al constructor de la **clase base** mediante la referencia `super`.

A la hora de destruir un objeto (método `finalize()`) es importante llamar a los finalizadores en el **orden inverso** a como fueron llamados los constructores (**primero se liberan los recursos de la clase derivada y después los de la clase base** mediante la llamada `super.finalize()`).

Si la clase `Persona` tuviera un constructor de este tipo:

```
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim) {  
    this.nombre= nombre;  
    this.apellidos= apellidos;  
    this.fechaNacim= new GregorianCalendar (fechaNacim);  
}
```

Podrías llamarlo desde un constructor de una clase derivada (por ejemplo `Alumno`) de la siguiente forma:

```
public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String grupo, double notaMedia) {  
    super (nombre, apellidos, fechaNacim);  
    this.grupo= grupo;  
    this.notaMedia= notaMedia;  
}
```

En realidad se trata de otro recurso más para optimizar la **reutilización de código**, en este caso el del **constructor**, que aunque no es heredado, sí puedes invocarlo para no tener que escribirlo.

### Autoevaluación

Puede invocarse al constructor de una superclase mediante el uso de la referencia `this`. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

La palabra reservada `this` es una referencia a la propia clase. Para hacer referencia a la superclase se utiliza la palabra reservada `super`.

## Ejercicio resuelto

Escribe un constructor para la clase `Profesor` que realice una llamada al constructor de su clase base para inicializar sus atributos heredados. Los atributos específicos (no heredados) sí deberán ser inicializados en el propio constructor de la clase `Profesor`.

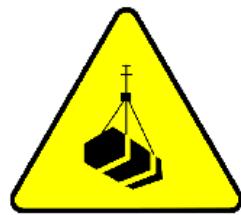
[Mostrar retroalimentación](#)

```
public Profesor (String nombre, String apellidos, GregorianCalendar fechaNacim, String especialidad){  
    super (nombre, apellidos, fechaNacim);  
    this.especialidad= especialidad;  
    this.salario= salario;  
}
```

## 3.7.- Creación y utilización de clases derivadas.

Ya has visto cómo crear una **clase derivada**, cómo acceder a los **miembros heredados** de las **clases superiores**, cómo redefinir algunos de ellos e incluso cómo invocar a un **constructor** de la **superclase**. Ahora se trata de poner en práctica todo lo que has aprendido para que puedas crear tus propias **jerarquías de clases**, o basarte en clases que ya existan en Java para heredar de ellas, y las utilices de manera adecuada para que tus aplicaciones sean más fáciles de escribir y mantener.

La idea de la **herencia** no es complicar los programas, sino todo lo contrario: **simplificarlos al máximo**. Procurar que haya que escribir la menor cantidad posible de código repetitivo e intentar facilitar en lo posible la realización de cambios (bien para corregir errores bien para incrementar la funcionalidad).



Ministerio de Educación y FP. ITE  
idITE=169339 (CC BY-NC-SA)

### Para saber más

Puedes echar un vistazo a este vídeo en el que se muestran algunos ejemplos de utilización de la herencia y clases derivadas. Está estructurado en tres partes:

<https://www.youtube.com/embed/Nu2ziz9Sq0g>

[Resumen textual alternativo](#)

## 3.8.- La clase Object en Java.

Todas las clases en Java son descendentes (directos o indirectos) de la clase `Object`. Esta clase define los **estados y comportamientos básicos que deben tener todos los objetos**. Entre estos comportamientos, se encuentran:

- ✓ La posibilidad de compararse.
- ✓ La capacidad de convertirse a cadenas.
- ✓ La habilidad de devolver la clase del objeto.



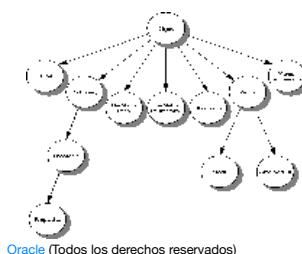
Diosdado Sánchez Hernández (CC BY-NC)

Entre los métodos que incorpora la clase `Object` y que por tanto hereda cualquier clase en Java tienes:

### Principales métodos de la clase Object

Método	Descripción
<code>Object ()</code>	Constructor.
<code>clone ()</code>	Método <b>clonador</b> : crea y devuelve una copia del objeto ("clona" el objeto).
<code>boolean equals (Object obj)</code>	Indica si el objeto pasado como parámetro es igual a este objeto.
<code>void finalize ()</code>	Método llamado por el <b>recolector de basura</b> cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
<code>int hashCode ()</code>	Devuelve un código <b>hash</b> para el objeto.
<code>toString ()</code>	Devuelve una representación del objeto en forma de <code>String</code> .

La clase `Object` representa la **superclase** que se encuentra en la cúspide de la **jerarquía de herencia** en Java. Cualquier clase (incluso las que tú implementes) acaban heredando de ella.



### Para saber más

Para obtener más información sobre la clase `Object`, sus métodos y propiedades, puedes consultar la documentación de la API de **Java** en el sitio web de Oracle.

[Documentación de la clase Object.](#)

### Autoevaluación

Toda clase Java tiene un método `toString` y un método `finalize`. ¿Verdadero o Falso?

- Verdadero  Falso

**Verdadero**

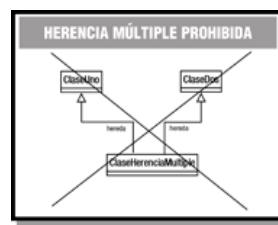
Dado que la clase `Object` tiene esos métodos y puesto que cualquier clase Java tiene como superclase (más o menos directa) a la clase `Object`, cualquier clase Java heredará esos métodos.

## 3.9.- Herencia múltiple.

En determinados casos podrías considerar la posibilidad de que se necesite heredar de más de una clase, para así disponer de los miembros de dos (o más) clases disjuntas (que no derivan una de la otra). La **herencia múltiple** permite hacer eso: recoger las distintas características (atributos y métodos) de clases diferentes formando una nueva clase derivada de varias clases base.

El problema en estos casos es la posibilidad que existe de que se produzcan ambigüedades, así, si tuviéramos miembros con el mismo identificador en clases base diferentes, en tal caso, ¿qué miembro se hereda? Para evitar esto, los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere utilizar un determinado miembro que pueda ser ambiguo.

Ahora bien, la posibilidad de **herencia múltiple** no está disponible en todos los lenguajes orientados a objetos, ¿lo estará en Java? La respuesta es negativa.



Ministerio de Educación y FP ([CC BY-NC](#))

En Java no existe la herencia múltiple de clases.

## 4.- Clases abstractas.

### Caso práctico

**María** está desarrollando nuevas clases para el proyecto de la **Clínica Veterinaria** y se ha dado cuenta de que, para aprovechar toda la potencia de la **herencia**, le vendría bien definir algunas clases que, sin embargo, nunca va a llegar a instanciar: – “¡Qué raro! Estoy definiendo una clase de la que nunca voy a tener objetos. Voy a instanciar a sus **subclases**, pero nunca a la **superclase**...”. – Piensa **María** en voz alta.– “¡No te preocupes! En realidad no es tan raro...”. – Le contesta **Juan**, que acaba de llegar.– “¿Ah no? ¿Tú crees?”. – “Totalmente. Es más habitual de lo que piensas. Son las **clases abstractas**. Y si has llegado tú misma a la conclusión de que las necesitas, mejor todavía, pues vas a entender con más facilidad su funcionamiento.”. – “¡Genial! Cuéntame más...”.



Ministerio de Educación y FP (CC BY-NC)

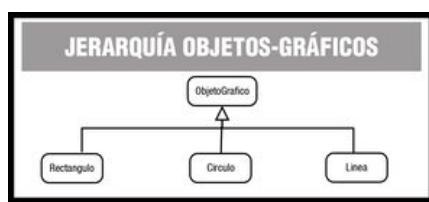
En determinadas ocasiones, es posible que necesites definir una clase que represente un concepto lo suficientemente abstracto como para que nunca vayan a existir instancias de ella (objetos). ¿Tendría eso sentido? ¿Qué utilidad podría tener?

Imagina una aplicación para un **centro educativo** que utilice las clases de ejemplo **Alumno** y **Profesor**, ambas subclases de **Persona**. Es más que probable que esa aplicación nunca llegue a necesitar objetos de la clase **Persona**, pues serían demasiado genéricos como para poder ser utilizados (no contendrían suficiente información específica). Podrías llegar entonces a la conclusión de que la clase **Persona** ha resultado de utilidad como **clase base** para construir otras clases que hereden de ella, pero no como una **clase instanciable** de la cual vayan a existir objetos. A este tipo de clases se les llama **clases abstractas**.

En algunos casos puede resultar útil disponer de clases que nunca serán instanciadas, sino que proporcionan un marco o modelo a seguir por sus clases derivadas dentro de una jerarquía de **herencia**. Son las **clases abstractas**.

La posibilidad de declarar **clases abstractas** es una de las características más útiles de los **lenguajes orientados a objetos**, pues permiten dar unas líneas generales de cómo es una clase sin tener que implementar todos sus métodos o implementando solamente algunos de ellos. Esto resulta especialmente útil cuando las distintas **clases derivadas** deban proporcionar los mismos métodos indicados en la clase **base abstracta**, pero su **implementación sea específica** para cada **subclase**.

Imagina que estás trabajando en un entorno de **manipulación de objetos gráficos** y necesitas trabajar con **líneas**, **círculos**, **rectángulos**, etc. Estos objetos tendrán en común algunos atributos que representen su estado (**ubicación**, **color del contorno**, **color de relleno**, etc.) y algunos métodos que modelen su comportamiento (**dibujar**, **llenar con un color**, **escalar**, **desplazar**, **rotar**, etc.). Algunos de ellos serán comunes para todos ellos (por ejemplo la **ubicación** o el **desplazamiento**) y sin embargo otros (como por ejemplo **dibujar**) necesitarán una implementación específica dependiendo del tipo de objeto. Pero, en cualquier caso, todos ellos necesitan esos métodos (tanto un **círculo** como un **rectángulo** necesitan el método **dibujar**, aunque se lleven a cabo de manera diferente). En este caso resultaría muy útil disponer de una **clase abstracta objeto gráfico** donde se definirían las **líneas generales** (algunos atributos concretos comunes, algunos métodos concretos comunes implementados y algunos métodos genéricos comunes sin implementar) de un objeto gráfico y más adelante, según se vayan definiendo **clases especializadas** (**líneas**, **círculos**, **rectángulos**), se irán concretando en cada **subclase** aquellos métodos que se dejaron sin implementar en la **clase abstracta**.



Ministerio de Educación y FP (CC BY-NC)

## Autoevaluación

Una clase abstracta no podrá ser nunca instanciada. ¿Verdadero o Falso?

- Verdadero  Falso

### Verdadero

Por definición una **clase abstracta** es una clase que no se puede instanciar. Nunca podrán existir objetos de una **clase abstracta**. La idea es proporcionar **un marco genérico** para otras subclases más especializadas que deriven de ella. Pero por sí misma no contiene suficiente información como para que existan instancias de ella.

## 4.1.- Declaración de una clase abstracta.

Ya has visto que una **clase abstracta** es una clase que no se puede instanciar, es decir, que no se pueden crear objetos a partir de ella. La idea es permitir que otras clases deriven de ella, proporcionando un **modelo genérico** y algunos **métodos de utilidad general**.



ITE idITE=110246 (CC BY-SA)

Las **clases abstractas** se declaran mediante el modificador `abstract`:

```
[modificador_acceso] abstract class nombreClase [herencia] [interfaces] {  
    ...  
}
```

Una clase puede contener en su interior métodos declarados como `abstract` (métodos para los cuales sólo se indica la cabecera, pero no se proporciona su implementación). En tal caso, la clase tendrá que ser necesariamente también `abstract`. Esos métodos tendrán que ser posteriormente implementados en sus **clases derivadas**.

Por otro lado, una clase también puede contener **métodos totalmente implementados (no abstractos)**, los cuales serán heredados por sus **clases derivadas** y podrán ser utilizados sin necesidad de definirlos (pues ya están implementados).

Cuando trabajes con **clases abstractas** debes tener en cuenta:

- ✓ Una **clase abstracta** sólo puede usarse para crear nuevas clases derivadas. No se puede hacer un `new` de una **clase abstracta**. Se produciría un **error de compilación**.
- ✓ Una **clase abstracta** puede contener **métodos totalmente definidos (no abstractos)** y **métodos sin definir (métodos abstractos)**.

### Autoevaluación

Puede llamarse al constructor de una clase abstracta mediante el operador `new`. ¿Verdadero o Falso?

Verdadero  Falso

**Falso**

Una **clase abstracta** no puede ser instanciada y por tanto no puede llamarse a su constructor mediante el uso de `new` para crear un nuevo objeto basado en ella (es en cierto modo una clase “sin terminar”). Se produciría un **error de compilación**.

### Ejercicio resuelto

Basándote en la jerarquía de clases de ejemplo (`Persona`, `Alumno`, `Profesor`), que ya has utilizado en otras ocasiones, modifica lo que consideres oportuno para que `Persona` sea, a partir de ahora, una clase abstracta (no instanciable) y las otras dos clases sigan siendo clases derivadas de ella, pero sí instanciables.

[Mostrar retroalimentación](#)

En este caso lo único que habría que hacer es añadir el modificador `abstract` a la clase `Persona`. El resto de la clase permanecería igual y las clases `Alumno` y `Profesor` no tendrían porqué sufrir ninguna modificación.

```
public abstract class Persona {  
    protected String nombre;  
    protected String apellidos;  
    protected GregorianCalendar fechaNacim;  
    ...  
}
```

A partir de ahora no podrán existir objetos de la clase `Persona`. El compilador generaría un **error**.

## 4.2.- Métodos abstractos.

Un **método abstracto** es un método cuya implementación no se define, sino que se declara únicamente su **interfaz** (cabecera) para que su cuerpo sea implementado más adelante en una **clase derivada**.

Un método se declara como abstracto mediante el uso del modificador **abstract** (como en las **clases abstractas**):

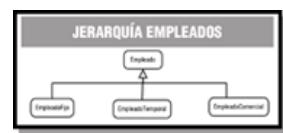
```
[modificador_acceso] abstract <tipo> <nombreMetodo> ([parámetros]) [excepciones];
```

Estos métodos tendrán que ser **obligatoriamente redefinidos** (en realidad “definidos”, pues aún no tienen contenido) en las **clases derivadas**. Si en una **clase derivada** se deja algún **método abstracto sin implementar**, esa **clase derivada** será también una **clase abstracta**.



Cuando una clase contiene un **método abstracto** tiene que declararse como **abstracta** obligatoriamente.

Imagina que tienes una clase **Empleado** genérica para diversos tipos de empleado y tres **clases derivadas**: **EmpleadoFijo** (tiene un salario fijo más ciertos complementos), **EmpleadoTemporal** (salario fijo más otros complementos diferentes) y **EmpleadoComercial** (una parte de salario fijo y unas comisiones por cada operación). La clase **Empleado** podría contener un **método abstracto calcularNomina**, pues sabes que se método será necesario para cualquier tipo de empleado (todo empleado cobra una nómina). Sin embargo el cálculo en sí de la nómina será diferente si se trata de un empleado fijo, un empleado temporal o un empleado comercial, y será dentro de las clases especializadas de **Empleado** (**EmpleadoFijo**, **EmpleadoTemporal**, **EmpleadoComercial**) donde se implementen de manera específica el cálculo de las mismas.



Ministerio de Educación y FP (CC BY-NC)

Debes tener en cuenta al trabajar con métodos abstractos:

- ✓ Un **método abstracto** implica que la clase a la que pertenece tiene que ser **abstracta**, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.
- ✓ Un **método abstracto** no puede ser **privado** (no se podría implementar, dado que las **clases derivadas** no tendrían acceso a él).
- ✓ Los **métodos abstractos** no pueden ser **estáticos**, pues los **métodos estáticos** no pueden ser redefinidos (y los **métodos abstractos** necesitan ser redefinidos).

### Autoevaluación

Los métodos de una clase abstracta tienen que ser también abstractos. ¿Verdadero o Falso?

Verdadero  Falso

Falso

Una **clase abstracta** puede contener **métodos abstractos**, pero también **métodos que estén completamente definidos (no abstractos)**. Lo que sí es cierto es que si una clase contiene algún **método abstracto**, entonces tendrá que ser obligatoriamente una **clase abstracta**.

### Ejercicio resuelto

Basándote en la jerarquía de clases **Persona**, **Alumno**, **Profesor**, crea un método abstracto llamado **mostrar** para la clase **Persona**. Dependiendo del tipo de persona (alumno o profesor) el método **mostrar** tendrá que mostrar unos u otros datos personales (habrá que hacer implementaciones específicas en cada clase derivada).

Una vez hecho esto, implementa completamente las tres clases (con todos sus atributos y métodos) y úsalas en un pequeño programa de ejemplo que cree un objeto de tipo Alumno y otro de tipo Profesor, los rellene con información y muestre esa información en la pantalla a través del método mostrar.

[Mostrar retroalimentación](#)

Dado que el método mostrar no va a ser implementado en la clase Persona, será declarado como **abstracto** y no se incluirá su implementación:

```
protected abstract void mostrar();
```

Recuerda que el simple hecho de que la clase **Persona** contenga un método abstracto hace que sea la clase sea abstracta (y deberá indicarse como tal en su declaración): `public abstract class Persona`.

En el caso de la clase Alumno habrá que hacer una implementación específica del método **mostrar** y lo mismo para el caso de la clase Profesor.

### 1. Método mostrar para la clase Alumno.

```
// Redefinición del método abstracto mostrar en la clase Alumno

public void mostrar() {
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
    System.out.printf ("Nombre: %s\n", this.nombre);
    System.out.printf ("Apellidos: %s\n", this.apellidos);
    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);
    System.out.printf ("Grupo: %s\n", this.grupo);
    System.out.printf ("Nota media: %.2f\n", this.notaMedia);
}
```

### 2. Método mostrar para la clase Profesor.

```
// Redefinición del método abstracto mostrar en la clase Profesor

public void mostrar() {
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
    System.out.printf ("Nombre: %s\n", this.nombre);
    System.out.printf ("Apellidos: %s\n", this.apellidos);
    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);
    System.out.printf ("Especialidad: %s\n", this.especialidad);
    System.out.printf ("Salario: %.2f euros\n", this.salario);
}
```

### 3. Programa de ejemplo de uso.

Un pequeño programa de ejemplo de uso del método **mostrar** en estas dos clases podría ser:

```

// Declaración de objetos

Alumno alumno;

Profesor profe;

// Creación de objetos (llamada a constructores)

alumno= new Alumno ("Juan", "Torres", new GregorianCalendar (1990, 10, 6), "1DAM-B", 7

profe= new Profesor ("Antonio", "Campos", new GregorianCalendar (1970, 8, 15), "Mates"

// Utilización del método mostrar

alumno.mostrar();

profesor.mostrar();

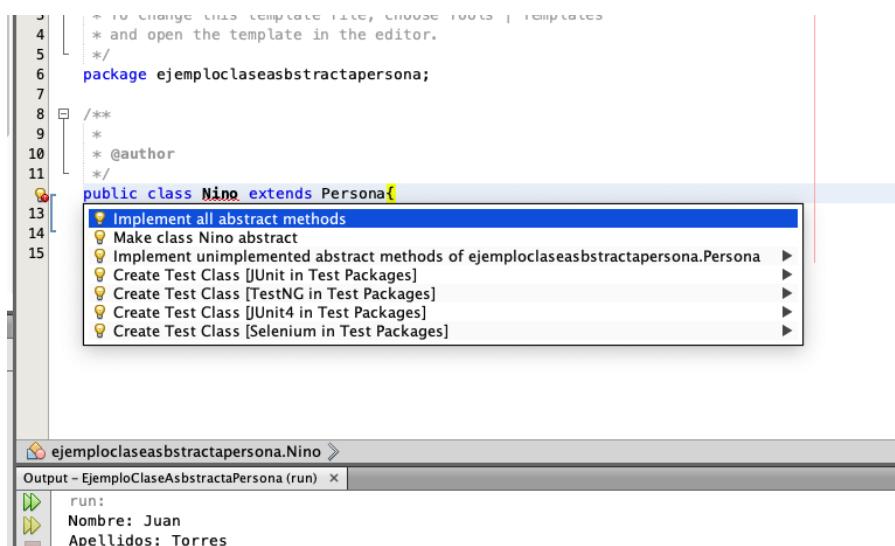
```

Puedes descargar del siguiente enlace un ejemplo completo en el que se declara la clase `Persona` como **abstracta** y con el **método abstracto** `mostrar`. Sus subclases `Alumno` y `Profesor` redefinen ese método especializándolo para cada una de ellas.

[Proyecto EjemploClaseAbstractaPersona.](#) (21 KB)

## Para saber más

Observa en la siguiente imagen cómo se pueden añadir los métodos abstractos heredados en una clase derivada de forma automática. Nuestro trabajo únicamente sería darles implementación porque de añadirlos ya se encarga Netbeans.



Observa cómo Netbeans sugiere varias opciones, entre ellas, declarar la clase **Niño** también abstracta.

## 4.3.- Clases y métodos finales.

En unidades anteriores has visto el modificador `final`, aunque sólo lo has utilizado por ahora para **atributos y variables** (por ejemplo para declarar **atributos constantes**, que una vez que toman un valor ya no pueden ser modificados). Pero este modificador también puede ser utilizado con clases y con métodos (con un comportamiento que no es exactamente igual, aunque puede encontrarse cierta analogía: no se permite heredar o no se permite redefinir).

**Una clase declarada como `final` no puede ser heredada**, es decir, **no puede tener clases derivadas**. La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):



Ministerio de Educación y FP (CC BY-NC-SA)

[modificador\_acceso] `final class` nombreClase [herencia] [interfaces]

Un **método** también puede ser declarado como `final`, en tal caso, ese método no podrá ser redefinido en una **clase derivada**:

[modificador\_acceso] `final <tipo> <nombreMetodo> ([parámetros]) [excepciones]`

Si intentas redefinir un método `final` en una subclase se producirá un **error de compilación**.

### Autoevaluación

Los modificadores `final` y `abstract` son excluyentes en la declaración de un **método**. ¿Verdadero o Falso?

- Verdadero  Falso

#### Verdadero

No se puede ser a la vez `final` y `abstract`, pues si un método `abstract` necesita ser redefinido, un método `final` no permite serlo. Sería una contradicción.

Además de en la declaración de atributos, clases y métodos, el modificador `final` también podría aparecer acompañando a un método de un parámetro. En tal caso no se podrá modificar el valor del parámetro dentro del código del método. Por ejemplo: `public final metodoEscribir (int par1, final int par2)`.

### Debes conocer

Dada la gran cantidad de contextos diferentes en los que se puede encontrar el modificador `final`, vale la pena que hagas un repaso de todos los lugares donde puede aparecer y cuál sería su función en cada uno. Puedes hacerlo en el [Anexo IV](#) de esta Unidad de Trabajo.

## 5.- Interfaces.

### Caso práctico

**María y Juan** continúan con su tarea de desarrollo de clases para el proyecto de la **Clínica Veterinaria**. Ya han utilizado la **herencia** en diversas ocasiones e incluso han escrito alguna **clase abstracta** para luego generar clases especializadas basadas en ella.

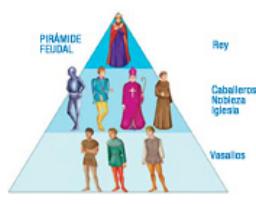
Pero ahora se les ha planteado un nuevo problema: tienen pensadas algunas clases entre las que no existe ninguna relación de **herencia**, cada una hereda de unos ancestros diferentes que no tienen nada que ver, pero, sin embargo, sí podrían compartir una buena parte de sus **comportamientos (métodos)**.

No es posible hacer que las dos hereden de la misma **clase base** porque hemos dicho que no se parecen en nada a ese respecto (cada una tiene su **clase base**, sin relación entre ellas), y tampoco pueden heredar de una nueva **clase abstracta** que contenga la interfaz de ese comportamiento, pues la **herencia múltiple** no está permitida en Java:

- "¿Qué hacemos entonces?, ¿repetimos la misma **interfaz** en las dos **jerarquías** de clases? No me cuadra tener que hacer eso...". - Le pregunta **María a Juan**.
- "A mí tampoco. No me suena muy bien". - Le contesta **Juan**.
- "Así es. Tiene que haber una solución más elegante que no nos haga tener que repetir ese código una y otra vez. ¿Alguna idea?".
- "Quizá exista una forma de resolver el problema. ¿Recuerdas que **Ada** nos habló el otro día de las **interfaces**?".



Ministerio de Educación y FP (CC BY-NC)



ITE idITE= 114183 (CC BY-SA)

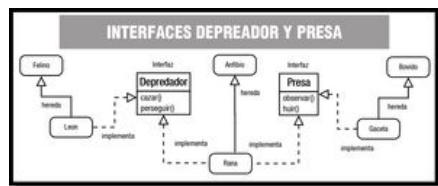
Has visto cómo la **herencia** permite definir **especializaciones** (o **extensiones**) de una **clase base** que ya existe sin tener que volver a repetir de todo el código de ésta. Este mecanismo da la oportunidad de que la nueva **clase especializada** (o **extendida**) disponga de toda la **interfaz** que tiene su clase base.

También has estudiado cómo los **métodos abstractos** permiten establecer una **interfaz** para marcar las **líneas generales de un comportamiento común de superclase** que deberían compartir de todas las **subclases**.

Si llevamos al límite esta idea de **interfaz**, podrías llegar a tener una **clase abstracta** donde todos sus métodos fueran abstractos. De este modo estarías dando únicamente el **marco de comportamiento**, sin ningún método implementado, de las posibles **subclases** que heredarán de esa **clase abstracta**. La idea de una **interfaz** (o **interface**) es precisamente ésa: **disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación** (no necesariamente jerárquica).

Una **interfaz** consiste principalmente en una lista de declaraciones de métodos sin implementar, que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, tendrá que implementar esos métodos establecidos en la **interfaz**. En este caso no se trata de una relación de **herencia** (la clase **A** es una especialización de la clase **B**, o la subclase **A** es del tipo de la superclase **B**), sino más bien una relación "de implementación de comportamientos" (la clase **A** implementa los métodos establecidos en la **interfaz B**, o los comportamientos indicados por **B** son llevados a cabo por **A**; pero no que **A** sea de clase **B**).

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieras que lleven a cabo están relacionadas con el hecho de que algunos animales sean **depredadores** (por ejemplo: **observar** una **presa**, **perseguirla**, **comérsela**, etc.) o sean **presas** (**observar**, **huir**, **esconderse**, etc.). Si creas la clase **León**, esta clase podría implementar una interfaz **Depredador**, mientras que otras clases como **Gacela** implementarían las acciones de la interfaz **Presa**. Por otro lado, podrías tener también el caso de la clase **Rana**, que implementaría las acciones de la interfaz **Depredador** (pues es cazador de pequeños insectos), pero también la de **Presa** (pues puede ser cazado y necesita las acciones necesarias para protegerse).



Ministerio de Educación y FP ([CC BY-NC](#))

## 5.1.- Concepto de interfaz.



ITE idITE= 134104 (CC BY-NC-SA)

Una **interfaz** en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.

Estos métodos sin implementar indican un **comportamiento**, un tipo de conducta, aunque no especifican cómo será ese **comportamiento (implementación)**, pues eso dependerá de las características específicas de cada clase que decida implementar esa **interfaz**. Podría decirse que una **interfaz** se encarga de establecer qué **comportamientos** hay que tener (qué **métodos**), pero no dice nada de cómo deben llevarse a cabo esos **comportamientos (implementación)**. Se indica sólo la **forma**, no la **implementación**.



ITE\_ Félix Vallés Calvo. idITE= 152206 (CC BY-NC-SA)

En cierto modo podrías imaginar el concepto de **interfaz** como un **guión** que dice: "éste es el protocolo de comunicación que deben presentar todas las clases que implementen esta interfaz". Se proporciona una lista de **métodos públicos** y, si quieras dotar a tu clase de esa **interfaz**, tendrás que definir todos y cada uno de esos **métodos públicos**.



ITE\_ Alessandro Quisi idITE= 123492 (CC BY-NC-SA)

En conclusión: **una interfaz se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es (de eso se encarga la clase y sus superclases), sino acciones (capacidades) que el objeto debería ser capaz de realizar**. Es por esto que el nombre de muchas interfaces en Java termina con sufijos del tipo "-able", "-or", "-ente" y cosas del estilo, que significan algo así como **capacidad o habilidad para hacer o ser receptores de algo (configurable, serializable, modificable, clonable, ejecutable, administrador, servidor, buscador, etc.)**, dando así la idea de que se tiene la capacidad de llevar a cabo el conjunto de acciones especificadas en la **interfaz**.

Imagínate por ejemplo la clase **Coche**, **subclase de Vehículo**. Los coches son **vehículos a motor**, lo cual implica una serie de acciones como, por ejemplo, **arrancar el motor** o **detener el motor**. Esa acción no la puedes heredar de **Vehículo**, pues no todos los vehículos tienen porqué ser a motor (piensa por ejemplo en una clase **Bicicleta**), y no puedes heredar de otra clase pues ya heredas de **Vehículo**. Una solución podría ser crear una **interfaz Arrancable**, que proporcione los métodos típicos de un **objeto a motor** (no necesariamente vehículos). De este modo la clase **Coche** sigue siendo subclase de **Vehículo**, pero también implementaría los comportamientos de la interfaz **Arrancable**, los cuales podrían ser también implementados por otras clases, hereden o no de **Vehículo** (por ejemplo una clase **Motocicleta** o bien una clase **Motosierra**). La clase **Coche** implementará su método **arrancar** de una manera, la clase **Motocicleta** lo hará de otra (aunque bastante parecida) y la clase **Motosierra** de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método **arrancar** como parte de la interfaz **Arrancable**.



ITE\_idITE=125306 (CC BY-NC-SA)

Según esta concepción, podrías hacerte la siguiente pregunta: **¿podrá una clase implementar varias interfaces?** La respuesta en este caso sí es afirmativa.

Una clase puede adoptar distintos modelos de comportamiento establecidos en diferentes interfaces. Es decir una clase puede implementar varias interfaces.

### Autoevaluación

**Una interfaz en Java no puede contener la implementación de un método mientras que una clase abstracta sí. ¿Verdadero o Falso?**

- Verdadero  Falso

#### Verdadero

Así es. Una interfaz no puede definir métodos (no implementa su contenido), tan solo los declara. Sin embargo en una clase abstracta se pueden dejar algunos métodos sin implementar (métodos abstractos) e implementar otros.

## 5.1.1.- ¿Clase abstracta o interfaz?

Observando el concepto de **interfaz** que se acaba de proponer, podría caerse en la tentación de pensar que es prácticamente lo mismo que una **clase abstracta** en la que **todos sus métodos sean abstractos**.

Es cierto que en ese sentido existe un gran **parecido formal** entre una **clase abstracta** y una **interfaz**, pudiéndose en ocasiones utilizar indistintamente una u otra para obtener un mismo fin. Pero, a pesar de ese gran parecido, existen algunas diferencias, no sólo formales, sino también conceptuales, muy importantes:



ITE\_idITE=113917 (CC BY-NC-SA)

- ✓ Una clase no puede heredar de varias clases, aunque sean abstractas (**herencia múltiple**). Sin embargo sí puede implementar una o varias **interfaces** y además seguir heredando de una clase.
- ✓ Una interfaz no puede definir métodos (**no implementa su contenido**), tan solo los declara o enumera.
- ✓ Una interfaz puede hacer que dos clases tengan un mismo **comportamiento** independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
- ✓ Una interfaz permite establecer un **comportamiento de clase sin apenas dar detalles**, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la **interfaz**).
- ✓ Las **interfaces tienen su propia jerarquía**, diferente e independiente de la jerarquía de clases.

De todo esto puede deducirse que una **clase abstracta** proporciona una **interfaz disponible sólo a través de la herencia**. Sólo quien herede de esa **clase abstracta** dispondrá de esa **interfaz**. Si una clase no pertenece a esa misma jerarquía (no hereda de ella) no podrá tener esa **interfaz**. Eso significa que para poder disponer de la **interfaz** podrías:

1. Volver a escribirla para esa jerarquía de clases. Lo cual no parece una buena solución.
2. Hacer que la clase herede de la superclase que proporciona la **interfaz** que te interesa, sacándola de su jerarquía original y convirtiéndola en **clase derivada** de algo de lo que conceptualmente no debería ser una **subclase**. Es decir, estarías forzando una relación "**es un**" cuando en realidad lo más probable es que esa relación no exista. Tampoco parece la mejor forma de resolver el problema.



ITE (CC BY-NC-SA)

Sin embargo, una **interfaz sí puede ser implementada por cualquier clase**, permitiendo que clases que no tengan ninguna relación entre sí (pertenezcan a distintas jerarquías) puedan compartir un determinado comportamiento (una interfaz) sin tener que forzar una relación de herencia que no existe entre ellas.

A partir de ahora podemos hablar de otra posible relación entre clases: la de **compartir un determinado comportamiento (interfaz)**. Dos clases podrían tener en común un determinado conjunto de comportamientos sin que necesariamente exista una relación jerárquica entre ellas. Tan solo cuando haya realmente una relación de tipo "**es un**" se producirá **herencia**.

### Recomendación

Si sólo vas a proporcionar una lista de **métodos abstractos (interfaz)**, sin definiciones de métodos ni atributos de objeto, suele ser recomendable definir una **interfaz** antes que **clase abstracta**. Es más, cuando vayas a definir una supuesta **clase base**, puedes comenzar declarándola como **interfaz** y sólo cuando veas que necesitas definir métodos o variables miembro, puedes entonces convertirla en **clase abstracta** (no instanciable) o incluso en una **clase instanciable**.

### Autoevaluación

En Java una clase no puede heredar de más de una clase abstracta ni implementar más de una interfaz.  
¿Verdadero o Falso?

Verdadero  Falso

Falso

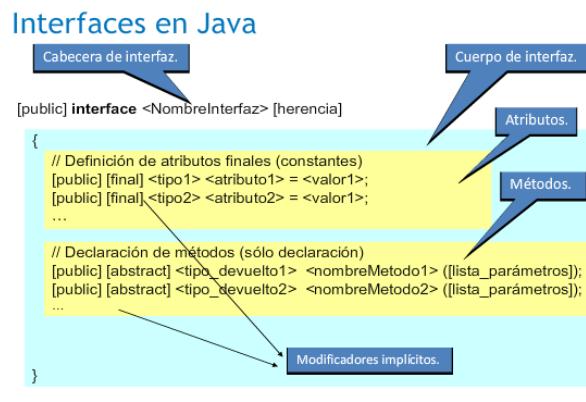
No. Aunque es cierto que solamente se puede heredar de una clase (sea abstracta o no), una clase Java

↳ Sí puede implementar varias interfaces (tantas como se quiera).

## 5.2.- Definición de interfaces.

La **declaración de una interfaz** en Java es similar a la declaración de una clase, aunque con algunas variaciones:

- ✓ Se utiliza la palabra reservada `interface` en lugar de `class`.
- ✓ Puede utilizarse el modificador `public`. Si incluye este modificador la **interfaz** debe tener el mismo nombre que el archivo `.java` en el que se encuentra (exactamente igual que sucedía con las clases). Si no se indica el modificador `public`, el acceso será por omisión o "de paquete" (como sucedía con las clases).
- ✓ Todos los **miembros** de la **interfaz** (atributos y métodos) son `public` de manera implícita. No es necesario indicar el modificador `public`, aunque puede hacerse.
- ✓ Todos los **atributos** son de tipo `final` y `public` (tampoco es necesario especificarlo), es decir, **constantes y públicos**. Hay que darles un **valor inicial**.
- ✓ Todos los **métodos** son **abstractos** también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.



Ministerio de Educación y FP ([CC BY-NC](#))

Como puedes observar, una **interfaz** consiste esencialmente en una lista de **atributos finales (constantes)** y **métodos abstractos (sin implementar)**. Su sintaxis quedaría entonces:

```
[public] interface <NombreInterfaz> {  
    [public] [final] <tipo1> <atributo1>= <valor1>;  
    [public] [final] <tipo2> <atributo2>= <valor2>;  
    ...  
    [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);  
    [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);  
    ...  
}
```

Si te fijas, la declaración de los métodos termina en punto y coma, pues no tienen cuerpo, al igual que sucede con los **métodos abstractos** de las **clases abstractas**.

El ejemplo de la interfaz **Depredador** que hemos visto antes podría quedar entonces así:

```
public interface Depredador {  
    void localizar (Animal presa);  
    void cazar (Animal presa);  
    ...  
}
```



[ajipodpics \(CC BY-SA\)](#)

Serán las clases que implementen esta interfaz (**León**, **Leopardo**, **Cocodrilo**, **Rana**, **Lagarto**, **Hombre**, etc.) las que definan

cada uno de los métodos por dentro.

## Autoevaluación

Los métodos de una interfaz en Java tienen que ser obligatoriamente declarados como `public` y `abstract`. Si no se indica así, se producirá un error de compilación. ¿Verdadero o Falso?

- Verdadero  Falso

### Falso

No es necesario hacerlo. Todos los métodos de una interfaz serán siempre `public` y `abstract` aunque no se indique. El compilador lo considerará así de forma implícita. Lo que sí produciría un error de compilación sería la colocación de un modificador de acceso diferente a `public` (por ejemplo `private` o `protected`) o el empleo del modificador `static`. Pero si no se indica nada, el método será `public` y `abstract`.

## Ejercicio resuelto

Crea una interfaz en Java cuyo nombre sea **Imprimible** que contenga un método útil para mostrar el contenido de una clase:

1. Método **devolverContenidoString**, que crea un `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves: "{<nombre\_atributo\_1>=<valor\_atributo\_1>, ..., <nombre\_atributo\_n>=<valor\_atributo\_n>}".

[Mostrar retroalimentación](#)

Se trata simplemente de declarar la interfaz e incluir en su interior ese método:

```
public interface Imprimible {  
    String devolverContenidoString();  
}
```

El cómo se implementará ese método dependerá exclusivamente de cada clase que decida implementar esta interfaz.

## 5.3.- Implementación de interfaces.

Como ya has visto, todas las clases que implementan una determinada **interfaz** están obligadas a proporcionar una **definición (implementación) de los métodos de esa interfaz**, adoptando el modelo de comportamiento propuesto por ésta.

Dada una **interfaz**, cualquier clase puede especificar dicha **interfaz** mediante el mecanismo denominado **implementación de interfaces**. Para ello se utiliza la palabra reservada `implements`:

```
class NombreClase implements NombreInterfaz {
```

De esta manera, la clase está diciendo algo así como "**la interfaz indica los métodos que debo implementar, pero voy a ser yo (la clase) quien los implemente**".

Es posible indicar varios nombres de **interfaces** separándolos por comas:

```
class NombreClase implements NombreInterfaz1, NombreInterfaz2, ... {
```

Cuando una clase implementa una **interfaz**, tiene que redefinir sus métodos nuevamente con **acceso público**. Con otro tipo de acceso se producirá un **error de compilación**. Es decir, que del mismo modo que no se podían restringir permisos de acceso en la **herencia de clases**, tampoco se puede hacer en la **implementación de interfaces**.

Una vez implementada una **interfaz** en una clase, los métodos de esa interfaz tienen exactamente el mismo tratamiento que cualquier otro método, sin ninguna diferencia, pudiendo ser invocados, heredados, redefinidos, etc.

En el ejemplo de los depredadores, al definir la clase **León**, habría que indicar que implementa la **interfaz Depredador**:

```
class Leon implements Depredador {
```

Y en su interior habría que implementar aquellos métodos que contenga la **interfaz**:

```
void localizar (Animal presa) {  
  
    // Implementación del método localizar para un león  
  
    ...  
  
}
```

En el caso de clases que pudieran ser a la vez **Depredador** y **Presa**, tendrían que implementar ambas interfaces, como podría suceder con la clase **Rana**:

```
class Rana implements Depredador, Presa {
```

Y en su interior habría que implementar aquellos métodos que contengan ambas **interfaces**, tanto las de **Depredador** (**localizar**, **cazar**, etc.) como las de **Presa** (**observar**, **hirir**, etc.).

### Autoevaluación

¿Qué palabra reservada se utiliza en Java para indicar que una clase va a definir los métodos indicados por una interfaz?

- implements.**
- uses.**

- extends.**
- Los métodos indicados por una **interfaz** no se definen en las clases pues sólo se pueden utilizar desde la propia **interfaz**.

¡Así es! ¡Enhorabuena! Se utiliza la palabra reservada **implements** seguida de las **interfaces** que se van a implementar separadas por comas.

Incorrecto No existe esa palabra reservada en Java.

¡Has fallado! Esa palabra reservada se utiliza para indicar que una clase **hereda** (es **subclase**) de otra.

¡No es así! Los métodos incluidos en una **interfaz** han de ser definidos por aquellas clases que **implementen** o lleven a cabo esa interfaz.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## Recomendación

Para añadir la implementación de los métodos de una clase que implementa una interfaz, podemos utilizar la funcionalidad de Netbeans de agregarlos de forma automática exactamente igual que hicimos con los métodos abstractos. Agiliza la inserción de código pues nos ahorraremos escribir la cabecera de todos los métodos a los que tenemos que dar implementación.

## Ejercicio resuelto

Haz que las clases **Alumno** y **Profesor** implementen la interfaz **Imprimible** que se ha escrito en el ejercicio anterior.

[Mostrar retroalimentación](#)

La primera opción que se te puede ocurrir es pensar que en ambas clases habrá que indicar que implementan la interfaz **Imprimible** y por tanto definir los métodos que ésta incluye: `devolverContenidoString`, `devolverContenidoHashtable` y `devolverContenidoArrayList`.

Si las clases **Alumno** y **Profesor** no heredaran de la misma clase habría que hacerlo obligatoriamente así, pues no comparten **superclase** y precisamente para eso sirven las **interfaces**: para implementar determinados comportamientos que no pertenecen a la estructura jerárquica de herencia en la que se encuentra una clase (de esta manera, clases que no tienen ninguna relación de herencia podrían compartir interfaz).

Pero en este caso podríamos aprovechar que ambas clases sí son **subclases** de una misma **superclase** (heredan de la misma) y hacer que la interfaz **Imprimible** sea implementada directamente por la **superclase** (**Persona**) y de este modo ahorrarnos bastante código. Así no haría falta indicar explícitamente que **Alumno** y **Profesor** implementan la interfaz **Imprimible**, pues lo estarán haciendo de forma implícita al heredar de una clase que ya ha implementado esa interfaz (la clase **Persona**, que es padre de ambas).

Una vez que los métodos de la **interfaz** estén implementados en la clase **Persona**, tan solo habrá que redefinir o ampliar los métodos de la **interfaz** para que se adapten a cada **clase hija** específica (**Alumno** o **Profesor**), ahorrándonos tener que escribir varias veces la parte de código que obtiene los atributos genéricos de la clase **Persona**.

### 1. Clase Persona.

Indicamos que se va a implementar la interfaz **Imprimible**:

```
public abstract class Persona implements Imprimible {  
    ...
```

Definimos el método **devolverContenidoHashtable** a la manera de como debe ser implementado para la clase **Persona**. Podría quedar, por ejemplo, así:

```
public Hashtable devolverContenidoHashtable () {  
  
    // Creamos la Hashtable que va a ser devuelta  
  
    Hashtable contenido= new Hashtable ();  
  
    // Añadimos los atributos de la clase  
  
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");  
  
    String stringFecha= formatoFecha.format(this.fechaNacim.getTime());  
  
    contenido.put ("nombre", this.nombre);  
    contenido.put ("apellidos", this.apellidos);  
    contenido.put ("fechaNacim", stringFecha);  
  
    // Devolvemos la Hashtable  
  
    return contenido;  
}
```

Del mismo modo, definimos también el método **devolverContenidoArrayList**:

```
public ArrayList devolverContenidoArrayList () { ... }
```

Y por último el método **devolverContenidoString**:

```
public String devolverContenidoString () { ... }
```

### 2. Clase Alumno.

Esta clase hereda de la clase **Persona**, de manera que heredará los tres métodos anteriores. Tan solo habrá que redefinirlos para que, aprovechando el código ya escrito en la **superclase**, se añada la funcionalidad específica que aporta esta **subclase**.

```
public class Alumno extends Persona {
```

...

Como puedes observar no ha sido necesario incluir el `implements Imprimible`, pues el `extends Persona` lo lleva implícito dado que `Persona` ya implementaba ese **interfaz**. Lo que haremos entonces será llamar al método que estamos redefiniendo utilizando la referencia a la **superclase super**.

El método `devolverContenidoHashtable` podría quedar, por ejemplo, así:

```
public Hashtable devolverContenidoHashtable () {  
    // Llamada al método de la superclase  
  
    Hashtable contenido= super.devolverContenidoHashtable();  
  
    // Añadimos los atributos específicos de la clase  
  
    contenido.put ("grupo", this.salario);  
  
    contenido.put ("notaMedia", this.especialidad);  
  
    // Devolvemos la Hashtable rellena  
  
    return contenido;  
}
```

### 3. Clase Profesor.

En este caso habría que proceder exactamente de la misma manera que con la clase `Alumno`: redefiniendo los métodos de la interfaz `Imprimible` para añadir la funcionalidad específica que aporta esta **subclase**.

Desde el siguiente enlace puedes descargar un ejemplo completo en el que se implementa la **interfaz Imprimible** en la clase `Persona` y cómo sus subclases `Alumno` y `Profesor` redefinen los métodos de esa interfaz heredada de `Persona`.

[Proyecto EjemploInterfazImprimible.](#)

## 5.4.- Simulación de la herencia múltiple mediante el uso de interfaces.

Una **interfaz** no tiene **espacio de almacenamiento** asociado (no se van a declarar objetos de un tipo de interfaz), es decir, no tiene **implementación**.

En algunas ocasiones es posible que interese representar la situación de que "una clase **X** es de tipo **A**, de tipo **B**, y de tipo **C**", siendo **A, B, C** **clases disjuntas** (no heredan unas de otras). Hemos visto que sería un caso de **herencia múltiple** que Java no permite.

Para poder simular algo así, podrías definir tres **interfaces A, B, C** que indiquen los comportamientos (métodos) que se deberían tener según se pertenezca a una supuesta clase **A, B, o C**, pero sin implementar ningún método concreto ni atributos de objeto (sólo interfaz).

De esta manera la clase **X** podría a la vez:

1. Implementar las interfaces **A, B, C**, que la dotarían de los comportamientos que deseaba heredar de las clases **A, B, C**.
2. Heredar de otra clase **Y**, que le proporcionaría determinadas características dentro de su taxonomía o jerarquía de objeto (atributos, métodos implementados y métodos abstractos).



Ministerio de Educación y FP [\(CC BY-NC\)](#)

En el ejemplo que hemos visto de las interfaces **Depredador** y **Presa**, tendrías un ejemplo de esto: la clase **Rana**, que es subclase de **Anfibio**, implementa una serie de **comportamientos** propios de un **Depredador** y, a la vez, otros más propios de una **Presa**. Esos **comportamientos** (métodos) no forman parte de la **superclase Anfibio**, sino de las **interfaces**. Si se decide que la clase **Rana** debe de llevar a cabo algunos otros **comportamientos adicionales**, podrían añadirse a una **nueva interfaz** y la clase **Rana** implementaría una tercera **interfaz**.

De este modo, con el mecanismo "**una herencia pero varias interfaces**", podrían conseguirse resultados similares a los obtenidos con la **herencia múltiple**.

Ahora bien, del mismo modo que sucedía con la **herencia múltiple**, puede darse el problema de la **colisión de nombres** al implementar dos **interfaces** que tengan un **método con el mismo identificador**. En tal caso puede suceder lo siguiente:

- ✓ Si los dos métodos tienen **diferentes parámetros** no habrá problema aunque tengan el mismo nombre pues se realiza una **sobrecarga** de métodos.
- ✓ Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se producirá un **error de compilación** (al igual que sucede en la sobrecarga cuando la única diferencia entre dos métodos es ésa).

Si los dos métodos son **exactamente iguales en identificador, parámetros y tipo devuelto**, entonces solamente se podrá **implementar uno de los dos métodos**. En realidad se trata de un solo método pues ambos tienen la misma interfaz (mismo identificador, mismos parámetros y mismo tipo devuelto).

### Recomendación

La utilización de nombres idénticos en diferentes **interfaces** que pueden ser implementadas a la vez por una misma clase puede causar, además del problema de la **colisión de nombres**, dificultades de **legibilidad** en el código, pudiendo dar lugar a confusiones. Si es posible intenta evitar que se produzcan este tipo de situaciones.

### Autoevaluación

**Dada una clase Java que implementa dos interfaces diferentes que contienen un método con el mismo nombre, indicar cuál de las siguientes afirmaciones es correcta.**

- Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se producirá un **error de compilación**.
- Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se **implementarán dos métodos**.

- Si los dos métodos son **exactamente iguales en identificador, parámetros y tipo devuelto**, se producirá un **error de compilación**.
- Si los dos métodos tienen **diferentes parámetros** se producirá un **error de compilación**.

¡Así es! ¡Enhorabuena! Se produce un **error de compilación** pues en la **sobrecarga** se utilizan los parámetros para distinguir unos métodos de otros, pero no el **valor de retorno**.

Incorrecto Se produce un **error de compilación** pues en la **sobrecarga** se utilizan los parámetros para distinguir unos métodos de otros, pero no el **valor de retorno**.

No es cierto. En estos casos solamente se podrá **implementar uno de los dos métodos**. En realidad se trata de un solo método pues ambos tienen la misma **interfaz** (mismo identificador, mismos parámetros y mismo tipo devuelto). No se producirá **ningún error de compilación**.

¡No es así! Si los dos métodos tienen **diferentes parámetros** no habrá problema pues se realiza una **sobrecarga** de métodos.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 5.5.- Herencia de interfaces.

Las **interfaces**, al igual que las **clases**, también permiten la **herencia**. Para indicar que una **interfaz** hereda de otra se indica nuevamente con la palabra reservada `extends`. Pero en este caso sí se permite la **herencia múltiple de interfaces**. Si se hereda de más de una **interfaz** se indica con la lista de **interfaces** separadas por comas.

Por ejemplo, dadas las interfaces **InterfazUno** e **InterfazDos**:

```
public interface InterfazUno {  
    // Métodos y constantes de la interfaz Uno  
}
```

```
public interface InterfazDos {  
    // Métodos y constantes de la interfaz Dos  
}
```

Podría definirse una nueva **interfaz** que heredara de ambas:

```
public interface InterfazCompleja extends InterfazUno, InterfazDos {  
    // Métodos y constantes de la interfaz compleja  
}
```

### Autoevaluación

En Java no está permitida la herencia múltiple ni para clases ni para interfaces. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

No. Aunque es cierto que no está permitida la **herencia múltiple** para **clases**, sí lo está para **interfaces**.

## 6.- Polimorfismo.

### Caso práctico

**María** está desarrollando algunas clases que representan categorías de **animales** para el proyecto de la **Clínica Veterinaria**. En algunos casos declara objetos de un tipo de animal y en ciertas ocasiones de otros, según las necesidades que tenga en cada momento. La clase **Animal** es demasiado genérica como para poder utilizarla en determinados casos y necesita clases más específicas para poder trabajar pues tendrá que usar unas u otras versiones de sus métodos. **Juan** también está haciendo algo parecido y ambos intuyen que el código que están escribiendo podría ser mucho más sencillo y flexible si pudieran declarar inicialmente objetos de la clase **Animal** y más tarde, durante la ejecución de la aplicación, utilizar objetos de tipo **Animal**, pero de clases más especializadas (**subclases de Animal**) en función de lo que suceda en cada momento. Sería muy interesante poder hacer algo así. **Ada** lleva algunos minutos escuchándolos y decide intervenir:



Ministerio de Educación y FP (CC BY-NC)

- "Veo que habéis llegado a la conclusión de que necesitáis trabajar con objetos cuya clase aún no está clara en tiempo de compilación, ¿no?". - Les pregunta a ambos.
- "Así es. Pero eso no se puede hacer, el compilador no nos lo va a permitir". - Le responden casi al unísono.
- "Bueno, es lógico que el compilador tenga que saber a qué clase pertenece un objeto para poder analizar si se está accediendo a los miembros correctos y con la sintaxis apropiada, ¿no crees?". - Les vuelve a preguntar.
- "Totalmente de acuerdo.". - Le contesta **María**.
- "Pero si declaramos un objeto de una clase que sea **superclase** de otras, quizás podríamos más tarde intentar para ese objeto instanciar una **subclase** más específica. Al fin y al cabo, una clase de tipo **MascotaDoméstica** sigue siendo también **Animal**, pues ha heredado de ella, ¿no es así?".
- "¿Quieres decir que podríamos utilizar en el programa objetos de clases cuyos métodos llamados no sabemos exactamente cuáles van a ser porque dependerá de la **subclase** concreta que se instancie en tiempo de ejecución?". - Le responde **María** muy interesada.
- "Parece que ha llegado el momento de que empiezis a trabajar con el polimorfismo y la ligadura dinámica". - Les contesta satisfecha.



El **polimorfismo** es otro de los grandes pilares sobre los que se sustenta la **Programación Orientada a Objetos** (junto con la **encapsulación** y la **herencia**). Se trata nuevamente de otra forma más de establecer diferencias entre interfaz e implementación, es decir, entre **el qué y el cómo**.

La **encapsulación** te ha permitido agrupar **características (atributos)** y **comportamientos (métodos)** dentro de una misma unidad (**clase**), pudiendo darles un mayor o menor componente de **visibilidad**, y permitiendo separar al máximo posible la **interfaz de la implementación**. Por otro lado la **herencia** te ha proporcionado la posibilidad de tratar a los objetos como pertenecientes a una **jerarquía de clases**. Esta capacidad va a ser fundamental a la hora de poder manipular muchos posibles objetos de clases diferentes como si fueran de la misma clase (**polimorfismo**).

Alessandro Pinna (CC BY-SA)

El **polimorfismo** te va a permitir mejorar la **organización** y la **legibilidad** del código así como la posibilidad de desarrollar aplicaciones que sean más fáciles de ampliar a la hora de incorporar nuevas funcionalidades. Si la implementación y la utilización de las clases es lo suficientemente genérica y extensible será más sencillo poder volver a este código para incluir nuevos requerimientos.

### Autoevaluación

¿Cuál de las siguientes características dirías que no es una de las que se suelen considerar como uno de los tres grandes pilares de la Programación Orientada a Objetos?

- Recursividad.
- Herencia.
- Polimorfismo.
- Encapsulación.

¡Así es! La **recursividad** no es una característica inherente al la **Programación Orientada a Objetos**. La incorporan muchos lenguajes no orientados a objetos.

No. La **herencia** (una **subclase** puede heredar miembros de una **superclase**) se considera una característica fundamental en la **Programación Orientada a Objetos**.

Error. Acabas de ver que **polimorfismo** se considera también una característica fundamental en la **Programación Orientada a Objetos**.

Has fallado. La **encapsulación** (agrupación de características y comportamiento en una misma unidad) se considera una característica fundamental en la **Programación Orientada a Objetos**.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 6.1.- Concepto de polimorfismo.

El **polimorfismo** consiste en la capacidad de poder utilizar una referencia a un objeto de una determinada clase como si fuera de otra clase (en concreto una **subclase**). Es una manera de decir que una clase podría tener varias (poli) formas (morfismo).

Un método "**polimórfico**" ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en **tiempo de ejecución** en lugar de en **tiempo de compilación**. Para poder hacer algo así es necesario utilizar métodos que pertenecen a una **superclase** y que en cada **subclase** se implementan de una forma en particular. En **tiempo de compilación** se invocará al método sin saber exactamente si será el de una subclase u otra (pues se está invocando al de la **superclase**). Sólo en **tiempo de ejecución** (una vez instanciada una u otra **subclase**) se conocerá realmente qué método (de qué **subclase**) es el que finalmente va a ser invocado.



dominiqueb (CC BY-NC-SA)

Esta forma de trabajar te va a permitir hasta cierto punto "desentenderte" del tipo de objeto **específico (subclase)** para centrarte en el tipo de objeto **genérico (superclase)**. De este modo podrás manipular objetos hasta cierto punto "desconocidos" en tiempo de compilación y que sólo durante la ejecución del programa se sabrá exactamente de qué tipo de objeto (**subclase**) se trata.

El **polimorfismo** ofrece la **posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclases**. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus **superclases**.

El **polimorfismo** puede llevarse a cabo tanto con **superclases** (abstractas o no) como con **interfaces**.

Dada una **superclase X**, con un método **m**, y dos **subclases A y B**, que redefinen ese método **m**, podrías declarar un objeto **O** de tipo **X** que en durante la **ejecución** podrá ser de tipo **A** o de tipo **B** (algo desconocido en **tiempo de compilación**). Esto significa que al invocarse el método **m** de **X (superclase)**, se estará en realidad invocando al método **m** de **A** o de **B** (alguna de sus **subclases**). Por ejemplo:

```
// Declaración de una referencia a un objeto de tipo X
ClaseX obj; // Objeto de tipo X (superclase)

...
// Zona del programa donde se instancia un objeto de tipo A (subclase) y se le asigna a la referencia obj.
// La variable obj adquiere la forma de la subclase A.
obj = ClaseA O;
...

// Otra zona del programa.

// Aquí se instancia un objeto de tipo B (subclase) y se le asigna a la referencia obj.
// La variable obj adquiere la forma de la subclase B.
obj = ClaseB O;
...

// Zona donde se utiliza el método m sin saber realmente qué subclase se está utilizando.

// (Sólo se sabrá durante la ejecución del programa)
obj.m() // Llamada al método m (sin saber si será el método m de A o de B).
...
```

Imagina que estás trabajando con las clases **Alumno** y **Profesor** y que en determinada zona del código podrías tener objetos, tanto de un tipo como de otro, pero eso sólo se sabrá según vaya discurriendo la ejecución del programa. En algunos casos, es posible que un determinado objeto pudiera ser de la clase **Alumno** y en otros de la clase **Profesor**, pero en cualquier caso serán objetos de la clase **Persona**. Eso significa que la llamada a un método de la clase **Persona** (por ejemplo

`devolverContenidoString`) en realidad será en unos casos a un método (con el mismo nombre) de la clase `Alumno` y, en otros, a un método (con el mismo nombre también) de la clase `Profesor`. Esto será posible hacerlo gracias a la **ligadura dinámica**.

## Autoevaluación

El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una clase A pueda tomar la forma de una referencia a un objeto de cualquier otra clase B. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

No es exactamente así. Para que eso pueda suceder la clase B debe ser subclase de A. En cualquier otro caso no funcionaría.

## 6.2.- Ligadura dinámica.

La conexión que tiene lugar durante una llamada a un método suele ser llamada **ligadura**, **vinculación** o **enlace** (en inglés **binding**). Si esta **vinculación** se lleva a cabo durante el proceso de compilación, se le suele llamar **ligadura estática** (también conocido como **vinculación temprana**). En los lenguajes tradicionales, no orientados a objetos, ésta es la única forma de poder resolver la **ligadura** (en **tiempo de compilación**). Sin embargo, en los **lenguajes orientados a objetos** existe otra posibilidad: la **ligadura dinámica** (también conocida como **vinculación tardía**, **enlace tardío** o **late binding**).



ITE\_Manuel Ayuso Sousa\_idITE= 127019 (CC BY-NC-SA)

La **ligadura dinámica** hace posible que sea el **tipo de objeto** instanciado (obtenido mediante el **constructor** finalmente utilizado para crear el objeto) y no el **tipo de la referencia** (el tipo indicado en la declaración de la variable que apuntará al objeto) lo que determine qué versión del método va a ser invocada. El **tipo de objeto** al que apunta la variable de tipo referencia sólo podrá ser conocido durante la **ejecución** del programa y por eso el **polimorfismo** necesita la **ligadura dinámica**.

En el ejemplo anterior de la clase **X** y sus **subclases A** y **B**, la llamada al método **m** sólo puede resolverse mediante ligadura dinámica, pues es imposible saber en tiempo de compilación si el método **m** que debe ser invocado será el definido en la subclase **A** o el definido en la subclase **B**:

```
// Llamada al método m (sin saber si será el método m de A o de B).  
obj.m () // Esta llamada será resuelta en tiempo de ejecución (ligadura dinámica)
```

### Ejercicio resuelto

Imagínate una clase que represente a **instrumento musical genérico (Instrumento)** y dos subclases que representen tipos de instrumentos específicos (por ejemplo **Flauta** y **Piano**). Todas las clases tendrán un método **tocarNota**, que será específico para cada subclase.

Haz un pequeño programa de ejemplo en Java que utilice el **polimorfismo** (referencias a la **superclase** que se convierten en instancias específicas de **subclases**) y la **ligadura dinámica** (llamadas a un método que aún no están resueltas en **tiempo de compilación**) con estas clases que representan instrumentos musicales. Puedes implementar el método **tocarNota** mediante la escritura de un mensaje en pantalla.

[Mostrar retroalimentación](#)

La clase **Instrumento** podría tener un único método (**tocarNota**):

```
public abstract class Instrumento {  
  
    public void tocarNota (String nota) {  
  
        System.out.printf ("Instrumento: tocar nota %s.\n", nota);  
  
    }  
  
}
```

En el caso de las clases **Piano** y **Flauta** puede ser similar, heredando de **Instrumento** y redefiniendo el método **tocarNota**:

```
public class Flauta extends Instrumento {  
  
    @Override  
  
    public void tocarNota (String nota) {
```

```

        System.out.printf ("Flauta: tocar nota %s.\n", nota);
    }

}

public class Piano extends Instrumento {

    @Override

    public void tocarNota (String nota) {

        System.out.printf ("Piano: tocar nota %s.\n", nota);

    }

}

```

A la hora de declarar una **referencia** a un objeto de tipo instrumento, utilizamos la **superclase** (**Instrumento**):

```
Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el **constructor** de alguna de sus **subclases** (**Piano**, **Flauta**, etc.):

```

if (<condición>) {

    // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)

    instrumento1= new Piano ();

}

else if (<condición>) {

// Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)

    instrumento1= new Flauta ();

} else {

    ...

}

```

Finalmente, a la hora de invocar el método **tocarNota**, no sabremos a qué versión (de qué **subclase**) de **tocarNota** se estará llamando, pues dependerá del tipo de objeto (**subclase**) que se haya instanciado. Se estará utilizando por tanto la **ligadura dinámica**:

```

// Interpretamos una nota con el objeto instrumento1

// No sabemos si se ejecutará el método tocarNota de Piano o de Flauta

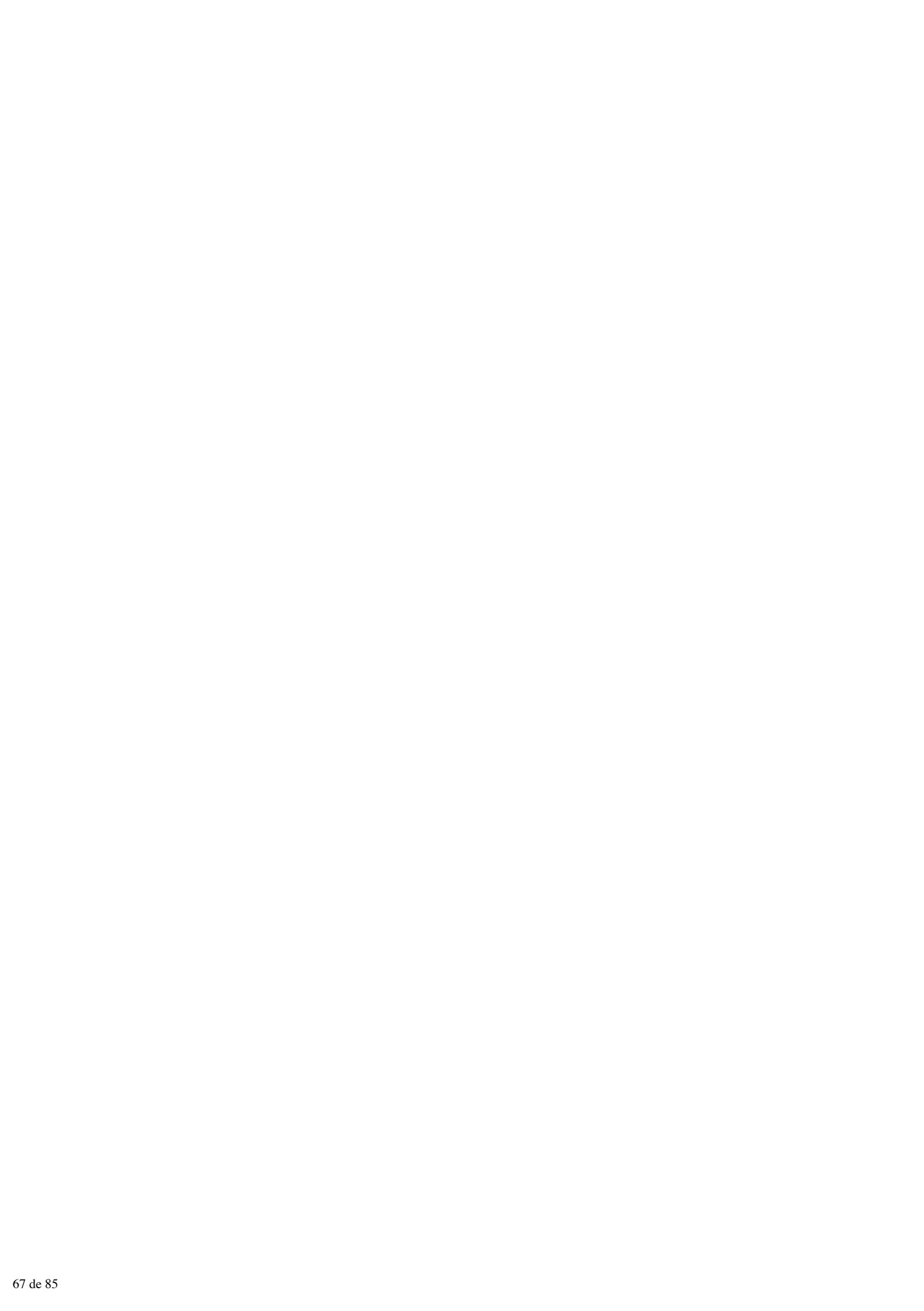
// (dependerá de la ejecución)

instrumento1.tocarNota ("do"); // Ejemplo de ligadura dinámica (tiempo de ejecución)

```

Desde el siguiente enlace puedes descargar el ejemplo completo en el que declaran las clases **Instrumento**, **Piano** y **Flauta** y son utilizadas para mostrar un ejemplo de **polimorfismo** y **ligadura dinámica**:

[Proyecto EjemploPolimorfismolnstrumentos.](#)



## 6.3.- Limitaciones de la ligadura dinámica.

Como has podido comprobar, el **polimorfismo** se basa en la utilización de **referencias** de un tipo más "amplio" (**superclases**) que los objetos a los que luego realmente van a apuntar (**subclases**). Ahora bien, existe una importante **restricción** en el uso de esta capacidad, pues el tipo de referencia limita cuáles son los métodos que se pueden utilizar y los atributos a los que se pueden acceder.

No se puede acceder a los **miembros específicos** de una **subclase** a través de una **referencia** a una **superclase**. Sólo se pueden utilizar los miembros declarados en la **superclase**, aunque la definición que finalmente se utilice en su ejecución sea la de la **subclase**.

Veamos un ejemplo: si dispones de una clase **A** que es subclase de **B** y declaras una variable como referencia un objeto de tipo **B**. Aunque más tarde esa variable haga referencia a un objeto de tipo **A** (**subclase**), los miembros a los que podrás acceder sin que el compilador produzca un error serán los miembros de **A** que hayan sido heredados de **B** (**superclase**). De este modo, se garantiza que los métodos que se intenten llamar van a existir cualquiera que sea la subclase de **B** a la que se apunte desde esa referencia.

En el ejemplo de las clases **Persona**, **Profesor** y **Alumno**, el **polimorfismo** nos permitiría declarar variables de tipo **Persona** y más tarde hacer con ellas referencia a objetos de tipo **Profesor** o **Alumno**, pero no deberíamos intentar acceder con esa variable a métodos que sean específicos de la clase **Profesor** o de la clase **Alumno**, tan solo a métodos que sabemos que van a existir seguro en ambos tipos de objetos (métodos de la **superclase Persona**).



ITE\_idITE= 153175 (CC BY-NC-SA)

### Ejercicio resuelto

Haz un pequeño programa en Java en el que se declare una variable de tipo **Persona**, se pidan algunos datos sobre esa persona (**nombre**, **apellidos** y si es **alumno** o si es **profesor**), y se muestren nuevamente esos datos en pantalla, teniendo en cuenta que esa variable no puede ser instanciada como un objeto de tipo **Persona** (es una **clase abstracta**) y que tendrás que instanciarla como **Alumno** o como **Profesor**. Recuerda que para poder recuperar sus datos necesitarás hacer uso de la **ligadura dinámica** y que tan solo deberías acceder a métodos que sean de la **superclase**.

[Mostrar retroalimentación](#)

Si tuviéramos diferentes variables referencia a objetos de las clases **Alumno** y **Profesor** tendrías algo así:

```
Alumno obj1;  
Profesor obj2:  
...  
// Si se dan ciertas condiciones el objeto será de tipo Alumno y lo tendrás en obj1  
System.out.printf ("Nombre: %s\n", obj1.getNombre());  
// Si se dan otras condiciones el objeto será de tipo Profesor y lo tendrás en obj2  
System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si pudieras tratar de una manera más genérica la situación, podrías intentar algo así:

```
Persona obj;  
// Si se dan ciertas condiciones el objeto será de tipo Alumno y por tanto lo instanciarás como tal  
obj = new Alumno (<parámetros>);
```

```
// Si se otras condiciones el objeto será de tipo Profesor y por tanto lo instanciarás como tal  
obj = new Profesor (<parámetros>);
```

De esta manera la variable `obj` podría contener una referencia a un objeto de la **superclase** `Persona` de **subclase** `Alumno` o bien de **subclase** `Profesor` (**polimorfismo**).

Esto significa que independientemente del tipo de **subclase** que sea (`Alumno` O `Profesor`), podrás invocar a métodos de la **superclase** `Persona` y durante la ejecución se resolverán como métodos de alguna de sus **subclases**:

```
//En tiempo de compilación no se sabrá de qué subclase de Persona será obj.  
//Habrá que esperar la ejecución para que el entorno lo sepa e invoque al método adecuado.  
System.out.printf ("Contenido del objeto usuario: %s\n", stringContenidoUsuario);
```

Por último recuerda que debes de proporcionar **constructores** a las **subclases** `Alumno` y `Profesor` que sean "compatibles" con algunos de los **constructores** de la **superclase** `Persona`, pues al llamar a un **constructor** de una **subclase**, su formato debe coincidir con el de algún **constructor** de la **superclase** (como debe suceder en general con cualquier método que sea invocado utilizando la **ligadura dinámica**).

Puedes descargar desde el siguiente enlace un ejemplo completo de uso del **polimorfismo** y la **ligadura dinámica** con las clases `Persona`, `Alumno` y `Profesor`:

[Proyecto EjemploPolimorfismoPersona](#)

## 6.4.- Interfaces y polimorfismo.

Es posible también llevar a cabo el **polimorfismo** mediante el uso de **interfaces**. Un objeto puede tener una referencia cuyo tipo sea una **interfaz**, pero para que el compilador te lo permita, la clase cuyo **constructor** se utilice para crear el objeto deberá implementar esa **interfaz** (bien por si misma o bien porque la implemente alguna **superclase**). Un objeto cuya referencia sea de tipo **interfaz** sólo puede utilizar aquellos métodos definidos en la **interfaz**, es decir, que no podrán utilizarse los atributos y métodos específicos de su clase, tan solo los de la **interfaz**.



pobre.ch (CC BY)

Las referencias de tipo **interfaz** permiten unificar de una manera bastante estricta la forma de utilizarse de objetos que pertenezcan a clases muy diferentes (pero que todas ellas implementan la misma **interfaz**). De este modo podrás hacer referencia a diferentes objetos que no tienen ninguna relación jerárquica entre sí utilizando la misma variable (referencia a la **interfaz**). Lo único que los distintos objetos tendrían en común es que implementan la misma **interfaz**. En este caso sólo podrás llamar a los métodos de la **interfaz** y no a los específicos de las clases.

Por ejemplo, si tenías una variable de tipo referencia a la interfaz **Arrancable**, podrías instanciar objetos de tipo **Coche** o **Motosierra** y asignarlos a esa referencia (teniendo en cuenta que ambas clases no tienen una relación de herencia). Sin embargo, tan solo podrás usar en ambos casos los métodos y los atributos de la interfaz **Arrancable** (por ejemplo **arrancar**) y no los de **Coche** o los de **Motosierra** (sólo los genéricos, nunca los específicos).

En el caso de las clases **Persona**, **Alumno** y **Profesor**, podrías declarar, por ejemplo, variables del tipo **Imprimible**:

```
Imprimible obj; // Imprimible es una interfaz y no una clase
```

Con este tipo de referencia podrás luego apuntar a objetos tanto de tipo **Profesor** como de tipo **Alumno**, pues ambos implementan la interfaz **Imprimible**:

```
// En algunas circunstancias podría suceder esto:  
obj= new Alumno (nombre, apellidos, fecha, grupo, nota); // Polimorfismo con interfaces  
...  
// En otras circunstancias podría suceder esto:  
obj= new Profesor (nombre, apellidos, fecha, especialidad, salario); // Polimorfismo con interfaces  
...
```

Y más adelante hacer uso de la **ligadura dinámica**:

```
// Llamadas sólo a métodos de la interfaz  
String contenido;  
contenido= obj.devolverContenidoString(); // Ligadura dinámica con interfaces
```

### Autoevaluación

El polimorfismo puede hacerse con referencias de superclases abstractas, superclases no abstractas o con interfaces. ¿Verdadero o Falso?

Verdadero  Falso

**Verdadero**

Así es. Pueden declararse variables de un tipo **superclase** (**abstracta** o no) y más tarde instanciarse objetos **subclase**, o bien declararse variables de un tipo **interfaz** y posteriormente instanciarse objetos de clases que implementen esa **interfaz**.

## 6.5.- Conversión de objetos.

Como ya has visto, en principio no se puede acceder a los **miembros específicos** de una **subclase** a través de una **referencia** a una **superclase**. Si deseas tener acceso a todos los métodos y atributos específicos del objeto **subclase** tendrás que realizar una **conversión explícita (casting)** que convierta la referencia más general (**superclase**) en la del tipo específico del objeto (**subclase**).

Para que puedas realizar conversiones entre distintas clases es obligatorio que exista una relación de **herencia** entre ellas (una debe ser clase derivada de la otra). Se realizará una **conversión implícita o automática** de **subclase** a **superclase** siempre que sea necesario, pues un objeto de tipo **subclase** siempre contendrá toda la información necesaria para ser considerado un objeto de la **superclase**.



ITE idITE=169555 (CC BY-NC)

Ahora bien, la conversión en sentido contrario (de **superclase** a **subclase**) debe hacerse de forma **explícita** y según el caso podría dar lugar a errores por falta de información (atributos) o de métodos. En tales casos se produce una **excepción** de tipo **ClassCastException**.

Por ejemplo, imagina que tienes una clase **A** y una clase **B**, **subclase** de **A**:

```
class ClaseA {  
    public int atrib1;  
}  
  
class ClaseB extends ClaseA {  
    public int atrib2;  
}
```

A continuación declaras una variable referencia a la clase **A (superclase)** pero sin embargo le asignas una referencia a un objeto de la clase **B (subclase)** haciendo uso del **polimorfismo**:

```
A obj; // Referencia a objetos de la clase A  
obj= new B (); // Referencia a objetos clase A, pero apunta realmente a objeto clase B (polimorfismo)
```

El objeto que acabas de crear como **instancia de la clase B (subclase de A)** contiene más información que la que la referencia **obj** te permite en principio acceder sin que el compilador genere un error (pues es de clase **A**). En concreto los objetos de la clase **B** disponen de **atrib1** y **atrib2**, mientras que los objetos de la clase **A** sólo de **atrib1**. Para acceder a esa información adicional de la clase especializada (**atrib2**) tendrás que realizar una **conversión explícita (casting)**:

```
// Casting del tipo A al tipo B (funcionará bien porque el objeto es realmente del tipo B)  
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib2);
```

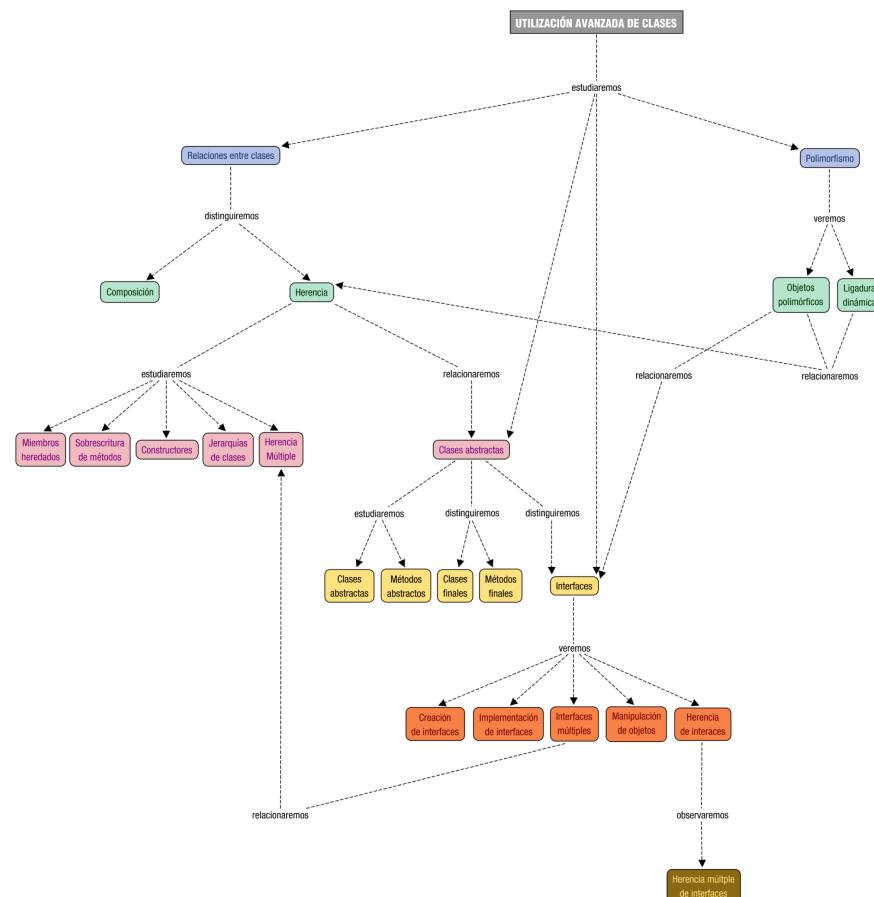
Sin embargo si se hubiera tratado de una **instancia de la clase A** y hubieras intentado acceder al miembro **atrib2**, se habría producido una **excepción** de tipo **ClassCastException**:

```
A obj; // Referencia a objetos de la clase A  
obj= new A (); // Referencia a objetos de la clase A, y apunta realmente a un objeto de la clase A  
// Casting del tipo A al tipo B (puede dar problemas porque el objeto es realmente del tipo A):  
// Funciona (la clase A tiene atrib1)  
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib1);  
// ¡Error en ejecución! (la clase A no tiene atrib2). Producirá una ClassCastException.
```

```
System.out.printf ("obj.attrib2=%d\n", ((B) obj).attrib2);
```

## 7.- Conclusiones.

Durante esta unidad de trabajo hemos trabajado con los dos conceptos más importantes del paradigma orientado a objetos: la herencia y el polimorfismo. Además, se ha introducido la relación de composición entre clases, mucho más sencillo a priori que las relaciones de herencia. Por otro lado, hemos trabajado con las clases abstractas y las interfaces. Todos los conceptos trabajados en esta unidad son fundamentales para abordar con garantías las siguientes puesto que todas las APIs incluidas en Java SE se basan en relaciones de herencia y pueden hacer uso del polimorfismo. Se pueden observar todos estos conceptos en el mapa conceptual de la unidad.



Ministerio de Educación y FP ([CC BY-NC](#))

En la siguiente unidad nos centraremos la entrada/salida de datos. Por lo tanto, conoceremos las clases que proporciona el API de Java para transferir datos a memoria secundaria.

# Anexo I.- Elaboración de los constructores de la clase Rectangulo.

---

## ENUNCIADO

Intenta describir los constructores de la clase `Rectangulo` teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase `Punto`, en lugar de cuatro elementos de tipo `double`):

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, `x1`, `y1`, `x2`, `y2`, que cree un rectángulo con los vértices (`x1`, `y1`) y (`x2`, `y2`).
3. Un constructor con dos parámetros, `punto1`, `punto2`, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, `base` y `altura`, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.

## POSIBLE SOLUCIÓN

Durante el proceso de creación de un objeto (**constructor**) de la **clase contenedora** (en este caso `Rectangulo`) hay que tener en cuenta también la creación (llamada a **constructores**) de aquellos objetos que son contenidos (en este caso objetos de la clase `Punto`).

En el caso del primer **constructor**, habrá que crear dos **puntos** con las coordenadas (0,0) y (1,1) y asignarlos a los atributos correspondientes (`vertice1` y `vertice2`):

```
public Rectangulo ()  
{  
    this.vertice1= new Punto (0,0);  
    this.vertice2= new Punto (1,1);  
}
```

Para el segundo **constructor** habrá que crear dos puntos con las coordenadas `x1`, `y1`, `x2`, `y2` que han sido pasadas como parámetros:

```
public Rectangulo (double x1, double y1, double x2, double y2)  
{  
    this.vertice1= new Punto (x1, y1);  
    this.vertice2= new Punto (x2, y2);  
}
```

En el caso del tercer **constructor** puedes utilizar directamente los dos puntos que se pasan como parámetros para construir los vértices del rectángulo:

Ahora bien, esto podría ocasionar un **efecto colateral** no deseado si esos objetos de tipo `Punto` son modificados en el futuro desde el código cliente del **constructor** (no sabes si esos puntos fueron creados especialmente para ser usados por el rectángulo o si pertenecen a otro objeto que podría modificarlos más tarde).

Por tanto, para este caso quizás sea recomendable crear dos nuevos puntos a imagen y semejanza de los puntos que se han pasado como parámetros. Para ello tendrías dos opciones:

1. Llamar al **constructor** de la clase `Punto` con los valores de los atributos (`x`, `y`).
2. Llamar al **constructor copia** de la clase `Punto`, si es que se dispone de él.

Aquí tienes las dos posibles versiones:

**Constructor** que “extrae” los atributos de los parámetros y crea nuevos objetos:

```

public Rectangulo (Punto vertice1, Punto vertice2)
{
    this.vertice1= vertice1;
    this.vertice2= vertice2;
}

```

Constructor que crea los nuevos objetos mediante el **constructor copia** de los parámetros:

```

public Rectangulo (Punto vertice1, Punto vertice2)
{
    this.vertice1= new Punto (vertice1.obtenerX(), vertice1.obtenerY());
    this.vertice2= new Punto (vertice2.obtenerX(), vertice2.obtenerY());
}

```

En este segundo caso puedes observar la utilidad de los **constructores de copia** a la hora de tener que **clonar** objetos (algo muy habitual en las inicializaciones).

Para el caso del **constructor** que recibe como parámetros la base y la altura, habrá que crear sendos vértices con valores (0,0) y (0 + base, 0 + altura), o lo que es lo mismo: (0,0) y (base, altura).

```

public Rectangulo (Punto vertice1, Punto vertice2)
{
    this.vertice1= new Punto (vertice1 );
    this.vertice2= new Punto (vertice2 );
}

```

Quedaría finalmente por implementar el **constructor copia**:

```

// Constructor copia

public Rectangulo (Rectangulo r) {
    this.vertice1= new Punto (r.obtenerVertice1());
    this.vertice2= new Punto (r.obtenerVertice2());
}

```

En este caso nuevamente volvemos a **clonar** los atributos `vertice1` y `vertice2` del objeto `r` que se ha pasado como parámetro para evitar tener que compartir esos atributos en los dos rectángulos.

## Anexo II.- Métodos para las clases heredadas Alumno y Profesor.

---

### ENUNCIADO

Dadas las clases `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos `get` y `set` en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos).

### POSIBLE SOLUCIÓN

#### 1. Clase `Alumno`.

Se trata de heredar de la clase `Persona` y por tanto utilizar con normalidad sus atributos heredados como si pertenecieran a la propia clase (de hecho se puede considerar que le pertenecen, dado que los ha heredado).

```
public class Alumno extends Persona {

    protected String grupo;

    protected double notaMedia;

    // Método getNombre

    public String getNombre (){

        return nombre;

    }

    // Método getApellidos

    public String getApellidos (){

        return apellidos;

    }

    // Método getFechaNacim

    public GregorianCalendar getFechaNacim (){

        return this.fechaNacim;

    }

    // Método getGrupo

    public String getGrupo (){

        return grupo;

    }

    // Método getNotaMedia

    public double getNotaMedia (){

        return notaMedia;

    }

    // Método setNombre

    public void setNombre (String nombre){

        this.nombre= nombre;

    }

    // Método setApellidos

    public void setApellidos (String apellidos){
```

```

    this.apellidos= apellidos;

}

// Método setFechaNacim

public void setFechaNacim (GregorianCalendar fechaNacim){

    this.fechaNacim= fechaNacim;

}

// Método setGrupo

public void setGrupo (String grupo){

    this.grupo= grupo;

}

// Método setNotaMedia

public void setNotaMedia (double notaMedia){

    this.notaMedia= notaMedia;

}

}

```

Si te fijas, puedes utilizar sin problema la referencia `this` a la propia clase con esos atributos heredados, pues pertenecen a la clase: `this.nombre`, `this.apellidos`, etc.

## 2. Clase Profesor.

Seguimos exactamente el mismo procedimiento que con la clase Alumno.

```

public class Profesor extends Profesor {

    String especialidad;

    double salario;

    // Método getNombre

    public String getNombre (){

        return nombre;

    }

    // Método getApellidos

    public String getApellidos (){

        return apellidos;

    }

    // Método getFechaNacim

    public GregorianCalendar getFechaNacim (){

        return this.fechaNacim;

    }

    // Método getEspecialidad

    public String getEspecialidad (){

        return especialidad;

    }
}

```

```

    }

    // Método getSalario

    public double getSalario (){
        return salario;
    }

    // Método setNombre

    public void setNombre (String nombre){
        this.nombre= nombre;
    }

    // Método setApellidos

    public void setApellidos (String apellidos){
        this.apellidos= apellidos;
    }

    // Método setFechaNacim

    public void setFechaNacim (GregorianCalendar fechaNacim){
        this.fechaNacim= fechaNacim;
    }

    // Método setSalario

    public void setSalario (double salario){
        this.salario= salario;
    }

    // Método setEspecialidad

    public void setEspecialidad (String especialidad){
        this.especialidad= especialidad;
    }
}

```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos **get** y **set** para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas heredaran también esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora. De esa manera te habrías evitado tener que escribir seis métodos en la clase `Alumno` y otros seis en la clase `Profesor`. Así que recuerda: **se pueden heredar tanto los atributos como los métodos**.

Aquí tienes un ejemplo de cómo podrías haber definido la clase `Persona` para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos):

```

public class Persona {

    protected String nombre;

    protected String apellidos;

    protected GregorianCalendar fechaNacim;

    // Método getNombre

```

```

public String getNombre (){
    return nombre;
}

// Método getApellidos

public String getApellidos (){
    return apellidos;
}

// Método getFechaNacim

public GregorianCalendar getFechaNacim (){
    return this.fechaNacim;
}

// Método setNombre

public void setNombre (String nombre){
    this.nombre= nombre;
}

// Método setApellidos

public void setApellidos (String apellidos){
    this.apellidos= apellidos;
}

// Método setFechaNacim

public void setFechaNacim (GregorianCalendar fechaNacim){
    this.fechaNacim= fechaNacim;
}

}

```

## Recomendación

Es fundamental conocer cómo se produce la herencia de métodos. Pero una vez entendido, la generación de estos métodos puede ser realizada de forma a automática por Netbeans a través de los mecanismos de generación automática de código. Además, Netbeans tienen en cuenta las particularidades de la herencia, es decir, no incluye métodos heredados en las clases derivadas. Aunque seguro que ya estás familiarizado con esta funcionalidad, puedes retroceder al punto 5.1 de la Unidad 5 para recordarlo, justo en la sección: Debes Conocer.

# Anexo III.- Métodos para los atributos de las clases Alumno y Profesor.

---

## ENUNCIADO

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos `get` y `set` en la clase `Persona` para trabajar con sus tres atributos y en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para `Persona` van a ser heredados en `Alumno` y en `Profesor`.

## POSIBLE SOLUCIÓN

### 1. Clase Persona.

```
public class Persona {  
  
    protected String nombre;  
  
    protected String apellidos;  
  
    protected GregorianCalendar fechaNacim;  
  
    // Método getNombre  
  
    public String getNombre (){  
  
        return nombre;  
    }  
  
    // Método getApellidos  
  
    public String getApellidos (){  
  
        return apellidos;  
    }  
  
    // Método getFechaNacim  
  
    public GregorianCalendar getFechaNacim (){  
  
        return this.fechaNacim;  
    }  
  
    // Método setNombre  
  
    public void setNombre (String nombre){  
  
        this.nombre= nombre;  
    }  
  
    // Método setApellidos  
  
    public void setApellidos (String apellidos){  
  
        this.apellidos= apellidos;  
    }  
  
    // Método setFechaNacim  
  
    public void setFechaNacim (GregorianCalendar fechaNacim){  
  
        this.fechaNacim= fechaNacim;  
    }  
}
```

## 2. Clase Alumno.

Al heredar de la clase `Persona` tan solo es necesario escribir métodos para los nuevos atributos (**métodos especializados** de acceso a los **atributos especializados**), pues los **métodos genéricos** (de acceso a los **atributos genéricos**) ya forman parte de la clase al haberlos heredado.

```
public class Alumno extends Persona {  
  
    protected String grupo;  
  
    protected double notaMedia;  
  
    // Método getGrupo  
  
    public String getGrupo (){  
  
        return grupo;  
    }  
  
    // Método getNotaMedia  
  
    public double getNotaMedia (){  
  
        return notaMedia;  
    }  
  
    // Método setGrupo  
  
    public void setGrupo (String grupo){  
  
        this.grupo= grupo;  
    }  
  
    // Método setNotaMedia  
  
    public void setNotaMedia (double notaMedia){  
  
        this.notaMedia= notaMedia;  
    }  
}
```

Aquí tienes una demostración práctica de cómo la herencia permite una reutilización eficiente del código, evitando tener que repetir atributos y métodos. Sólo has tenido que escribir cuatro métodos en lugar de diez.

## 3. Clase Profesor.

Seguimos exactamente el mismo procedimiento que con la clase `Alumno`.

```
public class Profesor extends Profesor {  
  
    String especialidad;  
  
    double salario;  
  
    // Método getEspecialidad  
  
    public String getEspecialidad (){  
  
        return especialidad;  
    }  
  
    // Método getSalario  
  
    public double getSalario (){
```

```
    return salario;  
}  
  
// Método setSalario  
  
public void setSalario (double salario){  
    this.salario= salario;  
}  
  
// Método setEspecialidad  
  
public void setEspecialidad (String especialidad){  
    this.especialidad= especialidad;  
}  
}
```

## Anexo IV.- Contextos del modificador final.

### Distintos contextos en los que puede aparecer el modificador final

Lugar	Función
Como modificador de clase.	La clase no puede tener subclases.
Como modificador de atributo.	El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.
Como modificador al declarar un método	El método no podrá ser redefinido en una clase derivada.
Como modificador al declarar una variable referencia.	Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.
Como modificador en un parámetro de un método.	El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Veamos un ejemplo de cada posibilidad:

#### 1. Modificador de una clase.

```
public final class ClaseSinDescendencia { // Clase "no heredable"  
...  
}
```

#### 2. Modificador de un atributo.

```
public class ClaseEjemplo {  
    // Valor constante conocido en tiempo de compilación  
    final double PI= 3.14159265;  
  
    // Valor constante conocido solamente en tiempo de ejecución  
    final int SEMILLA= (int) Math.random()*10+1;  
    ...  
}
```

#### 3. Modificador de un método.

```
public final metodoNoRedefinible (int parametro1) { // Método "no redefinible"  
    ...  
}
```

#### 4. Modificador en una variable referencia.

```
// Referencia constante: siempre se apuntará al mismo objeto Alumno recién creado,  
// aunque este objeto pueda sufrir modificaciones.  
  
final Alumno PRIMER_ALUMNO= new Alumno (“Pepe”, “Torres”, 9.55); // Ref. constante  
  
// Si la variable no es una referencia (tipo primitivo), sería una constante más  
// (como un atributo constante).  
  
final int NUMERO_DIEZ= 10; // Valor constante (dentro del ámbito de vida de la variable)
```

## 5. Modificador en un parámetro de un método.

```
void metodoConParametrosFijos (final int par1, final int par2) {  
  
    // Los parámetros “par1” y “par2” no podrán sufrir modificaciones aquí dentro  
  
    ...  
}
```

# Colecciones de Datos.

## Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

Ana sigue con la pequeña aplicación de procesamiento de pedidos que le pidió su jefa María.

En este caso, después de comprobar que el pedido tiene el formato correcto, algo que pensó hacer con expresiones regulares, llega el momento de almacenar en alguna estructura en memoria principal, si es posible de forma ordenada.

Ana ya conoce los arrays, como estructura de almacenamiento de datos en memoria pero ha llegado a la conclusión de que tienen ciertas limitaciones, sobre todo la imposibilidad de crecer en tiempo de ejecución y de mantener la estructura ordenada. Por otro lado, es bastante compleja la eliminación de elementos.

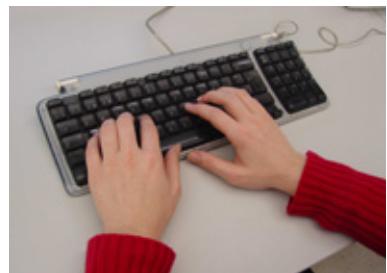
- Había pensado en utilizar arrays, es la única estructura de datos que conozco, pero hay limitaciones que no sé cómo solventar - comentó Ana. ¿Existirá alguna estructura de datos que no presente este tipo de limitaciones?-

- Si Ana, al igual que en otros lenguajes de programación, en Java se pueden utilizar estructuras de datos dinámicas. Pueden crecer en tiempo de ejecución tanto que las posibilidades tenga la memoria y además Java proporciona una serie de API para su manejo con mucha funcionalidad - le sugirió María.

- ¡Pues estoy deseando conocer esa API! -dijo Ana.

Vamos a ello, te sorprenderás de la cantidad de funcionalidad que está a disposición de los programadores.

Como ya estudiamos en una unidad anterior, cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?. ¿De qué herramientas disponemos cuando necesitamos almacenar muchos datos, tanto simples como compuestos?. La solución propuesta hasta el momento pasa por utilizar estructuras de datos definidas por el usuario, tal y como vimos en la unidad 6. Además, el programador debe desarrollar también los algoritmos que procesen esas estructuras de datos.



ITE. Óscar Javier Estupiñán Estupiñán.  
id=133827 ([CC BY-NC](#))

La mayoría de lenguajes de programación proporcionan dentro de su API conjuntos de clases e interfaces que liberan al programador de la necesidad de definir diferentes tipos de datos compuestos, como pueden ser listas, conjuntos, etc. En el caso de Java tenemos el [Framework Collection](#). Lo estudiaremos en profundidad a lo largo de esta unidad.



[Ministerio de Educación y Formación Profesional](#). (Dominio público)

**Materiales formativos de FP Online propiedad del Ministerio de  
Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Introducción a las colecciones.

## Caso práctico

A Ana las listas siempre se le han atragantado, por eso no las usa. Después de darle muchas vueltas, ha pensado que no le queda más remedio y que tendrá que usarlas para almacenar los artículos del pedido. Además, ha concluido que es la mejor forma de gestionar un grupo de objetos, aunque sean del mismo tipo.

No sabe si lo más adecuado es usar una lista u otro tipo de colección, así que ha decidido revisar todos los tipos de colecciones disponibles en Java, para ver cuál se adapta mejor a sus necesidades.



LatinStock ([CC BY-NC](#))

¿Qué consideras una colección? Pues seguramente al pensar en el término se te viene a la cabeza una colección de libros o algo parecido, y la idea no va muy desencaminada. **Una colección a nivel de software es un grupo de elementos almacenados de forma conjunta en una misma estructura.** Eso son las colecciones.



ITE idITE=109532 ([CC BY-NC](#))

Una colección o contenedor es un objeto que agrupa elementos múltiples en un objeto simple. Las colecciones son usadas para almacenar, recuperar y manipular datos.

Aunque un array o vector podría ser una colección, no está incluido en el framework Collections.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permiten manejar grupos de objetos, todo ello enfocado a potenciar la reusabilidad del software y facilitar las tareas de programación. Te parecerá increíble el tiempo que se ahorra empleando colecciones y cómo se reduce la complejidad del software usándolas adecuadamente. Las colecciones permiten almacenar y manipular grupos de objetos que, a priori, están relacionados entre sí (aunque no es obligatorio que estén relacionados, lo lógico es que si se almacenan juntos es porque tienen alguna relación entre sí), pudiendo trabajar con cualquier tipo de objeto (de ahí que se empleen los genéricos en las colecciones).

Además las colecciones permiten realizar algunas operaciones útiles sobre los elementos almacenados, tales como búsqueda u ordenación. En algunos casos es necesario que los objetos almacenados cumplan algunas condiciones (que implementen algunas interfaces), para poder ser utilizados por estos algoritmos.

Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación. En algunos lenguajes de programación su uso es algo más complejo

(como es el caso de C++), pero en Java su uso es bastante sencillo, es algo que descubrirás a lo largo de lo que queda de tema.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un modelo de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados, por lo que es necesario conocerlas. La interfaz inicial, a través de la cual se han construido el resto de colecciones, es la interfaz `java.util.Collection`, que define las operaciones comunes a todas las colecciones derivadas. A continuación se muestran las operaciones más importantes definidas por esta interfaz, ten en cuenta que `Collection` es una interfaz genérica donde "`<E>`" es el parámetro de tipo (podría ser cualquier clase):

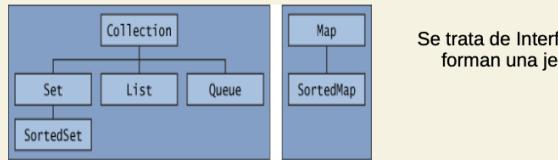
- ✓ Método `int size()`: retorna el número de elementos de la colección.
- ✓ Método `boolean isEmpty()`: retornará verdadero si la colección está vacía.
- ✓ Método `boolean contains (Object element)`: retornará verdadero si la colección tiene el elemento pasado como parámetro.
- ✓ Método `boolean add(E element)`: permitirá añadir elementos a la colección.
- ✓ Método `boolean remove (Object element)`: permitirá eliminar elementos de la colección.
- ✓ Método `Iterator<E> iterator()`: permitirá crear un iterador para recorrer los elementos de la colección. Esto se ve más adelante, no te preocupes.
- ✓ Método `Object[] toArray()`: permite pasar la colección a un array de objetos tipo `Object`.
- ✓ Método `containsAll(Collection<?> c)`: permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- ✓ Método `addAll (Collection<? extends E> c)`: permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
- ✓ Método `boolean removeAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- ✓ Método `boolean retainAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
- ✓ Método `void clear()`: vaciar la colección.

Más adelante veremos como se usan estos métodos, será cuando veamos las implementaciones (clases genéricas que implementan alguna de las interfaces derivadas de la interfaz `Collection`).

Los principales beneficios de utilizar este framework son:

1. **Reducen el esfuerzo de programación:** Disponen de un conjunto amplio de estructuras de datos y algoritmos, eficientes y probados.
2. **Incrementan la calidad de las aplicaciones.**
3. **Incrementan la interoperabilidad**, pues muchas APIs utilizan clases o interfaces de Collections como argumentos en sus métodos.
4. **Reducen el esfuerzo de aprender nuevas APIs.**
5. **Contribuyen a la reusabilidad del código.**

☞ **Core Collection Interfaces:** Contiene diferentes tipos de colecciones y suponen el fundamento del *Framework Collection*.



Todas las interfaces son genéricas → pueden almacenar cualquier tipo de datos.

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

Parámetro genérico → especificaremos al instanciar la colección el tipo de datos contenido.

## 2.- Clases y métodos genéricos (I).

### Caso práctico



ITE. Esther Balgoma Hernando. idITE=179681  
[\(CC BY-NC\)](#)

**María** se acerca a la mesa de **Ana**, quiere saber cómo lo lleva:

-¿Qué tal? ¿Cómo vas con la tarea? -pregunta María.

-Bien, creo. Mi programita ya sabe procesar el archivo de pedido y he creado un par de clases para almacenar los datos, pero no se como almacenar los artículos del pedido, porque son varios -comenta Ana.

-Pero, ¿cuál es el problema? Eso es algo sencillo.

-Pues que tengo crear un array para guardar con los artículos del pedido, y no se como averiguar el número de artículos antes de empezar a procesarlos. Es necesario saber el número de artículos para crear el array del tamaño adecuado.

-Pues en vez de utilizar un array, podrías utilizar una lista.

¿Sabes porqué se suele aprender el uso de los genéricos? Pues porque se necesita para usar las listas, aunque realmente los genéricos son una herramienta muy potente y que nos puede ahorrar tareas de programación repetitivas.

Las clases y los métodos genéricos son un recurso de programación disponible en muchos lenguajes de programación. Su objetivo es claro: facilitar la reutilización del software, creando métodos y clases que puedan trabajar con diferentes tipos de objetos, evitando incomodas y engorrosas conversiones de tipos. Su inicio se remonta a las plantillas (templates) de C++, un gran avance en el mundo de programación sin duda. En lenguajes de más alto nivel como Java o C# se ha transformado en lo que se denomina "genéricos". Veamos un ejemplo sencillo de como transformar un método normal en genérico:



Salvador Romero Villegas [\(CC BY-NC\)](#)

**Versión genérica y no genérica del método `compararTamano`.**

**Versión no Genérica**

```
public class util {
```

```

public static int compararTamano(Object[] a, Object[] b) {
    return a.length-b.length;
}
}

```

## Versión Genérica

```

public class util {

    public static <T> int compararTamano (T[] a, T[] b) {
        return a.length-b.length;
    }
}

```

Los dos métodos anteriores tienen un claro objetivo: permitir comprobar si un array es mayor que otro. Retornarán 0 si ambos arrays son iguales, un número mayor de cero si el array **b** es mayor, y un número menor de cero si el array **a** es mayor, pero uno es genérico y el otro no. La versión genérica del módulo incluye la expresión "<T>", justo antes del tipo returned por el método. "<T>" es la definición de una **variable o parámetro formal de tipo** de la clase o método genérico, al que podemos llamar simplemente **parámetro de tipo o parámetro genérico**. Este parámetro genérico (**T**) se puede usar a lo largo de todo el método o clase, dependiendo del ámbito de definición, y hará referencia a cualquier clase con la que nuestro algoritmo tenga que trabajar. Como veremos más adelante, puede haber más de un parámetro genérico.

Utilizar genéricos tiene claras ventajas. Para invocar un método genérico, sólo hay que realizar una **invocación de tipo genérico**, olvidándose de las conversiones de tipo. Esto consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico ("<T>"), para después, pasándole los argumentos correspondientes, ejecutar el algoritmo. Cada clase o interfaz concreta, la podemos denominar **tipo o tipo base** y se da por sentado que **los argumentos pasados al método genérico serán también de dicho tipo base**.

Supongamos que el tipo base es **Integer**, pues para realizar la invocación del método genérico anterior basta con indicar el tipo, entre los símbolos de menor qué y mayor qué ("<Integer>"), justo antes del nombre del método.

### Invocaciones de las versiones genéricas y no genéricas de un método.

Invocación del método no genérico.	Invocación del método genérico.
<pre> Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.compararTamano ((Object[])a, (Object[])b); </pre>	<pre> Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.&lt;Integer&gt;compararTamano (a, b); </pre>

## 2.1.- Clases y métodos genéricos (II).

¿Crees qué el código es más legible al utilizar genéricos o que se complica? La verdad es que al principio cuesta, pero después, el código se entiende mejor que si se empieza a insertar conversiones de tipo.

Las clases genéricas son equivalentes a los métodos genéricos pero a nivel de clase, permiten definir un parámetro de tipo o genérico que se podrá usar a lo largo de toda la clase, facilitando así crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase:



Berteun. (Dominio público)

```
public class Util<T> {  
  
    T t1;  
  
    public void invertir(T[] array) {  
  
        for (int i = 0; i < array.length / 2; i++) {  
  
            t1 = array[i];  
  
            array[i] = array[array.length - i - 1];  
  
            array[array.length - i - 1] = t1;  
  
        }  
  
    }  
  
}
```

En el ejemplo anterior, la clase `Util` contiene el método `invertir` cuya función es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre los símbolos menor que ("`<`") y mayor que ("`>`"), justo detrás del nombre de la clase. Veamos un ejemplo:

```
Integer[] numeros={0,1,2,3,4,5,6,7,8,9};  
  
Util<Integer> u= new Util<Integer>();  
  
u.invertir(numeros);  
  
for (int i=0;i<numeros.length;i++) System.out.println(numeros[i]);
```

Como puedes observar, el uso de genéricos es sencillo, tanto a nivel de clase como a nivel de método. Simplemente, a la hora de crear una instancia de una clase genérica, hay que especificar el tipo, tanto en la definición (`util <integer> u`) como en la creación (`new Util<Integer>()`).

Los genéricos los vamos a usar ampliamente a partir de ahora, aplicados a un montón de clases genéricas que tiene Java y que son de gran utilidad, por lo que es conveniente que aprendas bien a usar una clase genérica.

Los parámetros de tipo de las clases genéricas solo pueden ser clases, no pueden ser jamás tipos de datos primitivos como `int`, `short`, `double`, etc. En su lugar, debemos usar sus clases envoltorio `Integer`, `Short`, `Double`, etc.

## Autoevaluación

Dada la siguiente clase, donde el código del método prueba carece de importancia, ¿podrías decir cuál de las siguientes invocaciones es la correcta?

```
public class Util {  
  
    public static <T> int prueba (T t) { ... }  
  
};
```

- `Util.<int>prueba(4);`
- `Util.<Integer>prueba(new Integer(4));`
- `Util u=new Util(); u.<int>prueba(4);`

Incorrecto. No se pueden usar tipos de datos primitivos en los genéricos.

Correcto. Has captado la idea.

Incorrecto. Fíjate que `prueba` es un método estático y no se puede invocar así. Además se usan datos primitivos en un genérico, cosa que no es posible.

# Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

### 3.- Conjuntos (I).

## Caso práctico

Ana se toma un descanso, se levanta y en el pasillo se encuentra con Juan, con el que entabla una conversación bastante amena. Una cosa lleva a otra y al final, Ana saca el tema que más le preocupa:

—¿Cuántos tipos de colecciones hay? ¿Tu te los sabes? —pregunta Ana.

—¿Yo? ¡Que va! Normalmente consulto la documentación cuando los voy a usar, como todo el mundo. Lo que sí creo recordar es que había cuatro tipos básicos: los conjuntos, las listas, las colas y alguno más que no recuerdo. ¡Ah sí!, los mapas, aunque creo que no se consideraban un tipo de colección. ¿Por qué lo preguntas?

—Pues porque tengo que usar uno y no sé cuál.



Ministerio de Educación ([CC BY-NC](#))

¿Con qué relacionarías los conjuntos? Seguro que con las matemáticas. Los conjuntos son un tipo de colección que **no admite duplicados**, derivados del concepto matemático de conjunto.

La interfaz `java.util.Set` define cómo deben ser los conjuntos, y extiende la interfaz `Collection`, aunque no añade ninguna operación nueva. Las implementaciones (clases genéricas que implementan la interfaz `Set`) más usadas son las siguientes:

- ✓ `java.util.HashSet`. Conjunto que almacena los objetos usando ..... tablas hash, lo cual acelera enormemente el acceso a los objetos almacenado. Inconvenientes: necesitan bastante memoria y **no almacenan los objetos de forma ordenada** (al contrario pueden aparecer completamente desordenados).
- ✓ `java.util.LinkedHashSet`. Conjunto que almacena objetos combinando tablas hash, para un acceso rápido a los datos, y ..... listas enlazadas para conservar el orden. **El orden de almacenamiento es el de inserción**, por lo que se puede decir que es una estructura ordenada a medias. Inconvenientes: necesitan bastante memoria y es algo mas lenta que `HashSet`.
- ✓ `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores. pero tienen una gran ventaja: **los datos almacenados se ordenan por valor**. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno.



Fercufer ([CC BY-SA](#))

Poco a poco, iremos viendo que son las listas enlazadas y los árboles (no profundizaremos en los árboles rojo-negro, pero si veremos las estructuras tipo árbol en general). Veamos un ejemplo de

uso básico de la estructura `HashSet` y después, profundizaremos en los `LinkedHashSet` y los `TreeSet`.

Para crear un conjunto, simplemente creamos el `HashSet` indicando el tipo de objeto que va a almacenar, dado que es una clase genérica que puede trabajar con cualquier tipo de dato debemos crearlo como sigue (no olvides hacer la importación de `java.util.HashSet` primero):

```
HashSet<Integer> conjunto=new HashSet<Integer>();
```

Después podremos ir almacenando objetos dentro del conjunto usando el método `add` (definido por la interfaz `Set`). Los objetos que se pueden insertar serán siempre del tipo especificado al crear el conjunto:

```
Integer n=new Integer(10);

if (!conjunto.add(n)) System.out.println("Número ya en la lista.");
```

Si el elemento ya está en el conjunto, el método `add` retornará `false` indicando que no se pueden insertar duplicados. Si todo va bien, retornará `true`.

## Autoevaluación

¿Cuál de las siguientes estructuras ordena automáticamente los elementos según su valor?

- `HashSet`.
- `LinkedHashSet`.
- `TreeSet`.

No es correcto. Lee de nuevo los contenidos.

Incorrecto. Revisa el apartado y averigua por qué.

Efectivamente, has captado la idea a la primera.

## Solución

- 1. Incorrecto
  - 2. Incorrecto
  - 3. Opción correcta
-

## 3.1.- Conjuntos (II).

Y ahora te preguntarás, ¿cómo accedo a los elementos almacenados en un conjunto? Para obtener los elementos almacenados en un conjunto hay que usar iteradores, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (no hay otra forma de acceder a los elementos de un conjunto, es su inconveniente). Los iteradores se ven en mayor profundidad más adelante, de momento, vamos a usar iteradores de forma transparente, a través de una estructura `for` especial, denominada bucle "for-each" o bucle "para cada". En el siguiente código se usa un bucle for-each, en él la variable `i` va tomando todos los valores almacenados en el conjunto hasta que llega al último:

```
for (Integer i: conjunto) {  
  
    System.out.println("Elemento almacenado:"+i);  
  
}
```

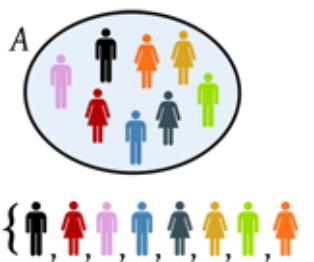


Ilustración de persona: AIIGA symbol signs collection; trabajo derivado por kismalac (CC BY-SA)

Como ves la estructura `for-each` es muy sencilla: la palabra `for` seguida de "(tipo variable:colección)" y el cuerpo del bucle; tipo es el tipo del objeto sobre el que se ha creado la colección, variable pues es la variable donde se almacenará cada elemento de la colección y colección pues la colección en sí. Los bucles `for-each` se pueden usar para todas las colecciones.

### Ejercicio resuelto

Realiza un pequeño programita que pregunte al usuario 5 números diferentes (almacenándolos en un `HashSet`), y que después calcule la suma de los mismos (usando un bucle `for-each`).

[Mostrar retroalimentación](#)

Una solución posible podría ser la siguiente. Para preguntar al usuario un número y para mostrarle la información se ha usado la clase `JOptionPane`, pero podrías haber utilizado cualquier otro sistema. Fijate en la solución y verás que el uso de conjuntos ha simplificado enormemente el ejercicio, permitiendo al programador o la programadora centrarse en otros aspectos:

```
public class EjemploHashSet {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        HashSet<Integer> conjunto=new HashSet<Integer>();
        String str;
        do {
            JOptionPanePanel.showInputDialog("Introduce un numero "+conjunto.size());
            try {
                Integer nu=Integer.parseInt(str);
                if (!conjunto.add(nu))
                    JOptionPanePanel.showMessageDialog(null, "Número ya es en la lista. Debes introducir otro.");
            } catch (NumberFormatException e) {
                JOptionPanePanel.showMessageDialog(null, "Número erroneo.");
            }
        } while (conjunto.size()<5);
        int suma;
        for (Integer il:conjunto) {
            suma+=il;
        }
        JOptionPanePanel.showMessageDialog(null,"La suma es:"+suma);
    }
}
```

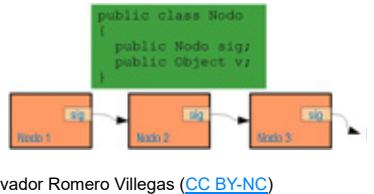
Salvador Romero Villegas y los autores de la herramienta de Software Libre NetBeans IDE 7.0 ([GNU/GPL](#))

## Ejemplo con HashSet

## 3.2.- Conjuntos (III).

¿En qué se diferencian las estructuras `LinkedHashSet` y `TreeSet` de la estructura `HashSet`? Ya se comentó antes, y es básicamente en su funcionamiento interno.

La estructura `LinkedHashSet` es una estructura que internamente funciona como una lista enlazada, aunque usa también tablas hash para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de la derecha, la cual está compuesta por nodos (elementos que forman la lista) que van enlazándose entre sí. Un nodo contiene dos cosas: el dato u objeto almacenado en la lista y el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (`null`) en la variable que contiene el siguiente nodo.

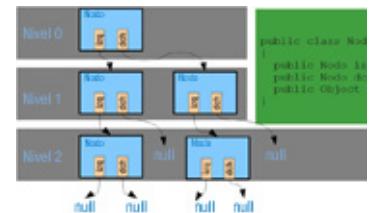


Salvador Romero Villegas ([CC BY-NC](#))

Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc. Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles de programación.

La estructura `TreeSet`, en cambio, utiliza internamente árboles. Los árboles son como las listas pero mucho más complejos. En vez de tener un único elemento siguiente, pueden tener dos o más elementos siguientes, formando estructuras organizadas y jerárquicas.

Los nodos se diferencian en dos tipos: nodos padre y nodos hijo; un nodo padre puede tener varios nodos hijo asociados (depende del tipo de árbol), dando lugar a una estructura que parece un árbol invertido (de ahí su nombre).



Salvador Romero Villegas ([CC BY-NC](#))

En la figura de la derecha se puede apreciar un árbol donde cada nodo puede tener dos hijos, denominados izquierdo (`izq`) y derecho (`dch`). Puesto que un nodo hijo puede también ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el primer nivel (que no tiene padre).

Los árboles son estructuras complejas de manejar y que permiten operaciones muy sofisticadas. Los árboles usados en los `TreeSet`, los árboles rojo-negro, son árboles auto-ordenados, es decir, que al insertar un elemento, este queda ordenado por su valor de forma que al recorrer el árbol, pasando por todos los nodos, los elementos salen ordenados. El ejemplo mostrado en la imagen es simplemente un árbol binario, el más simple de todos.

Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno (inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial. En la siguiente tabla tienes un uso comparado de `TreeSet` y `LinkedHashSet`. Su creación es similar a como se hace con `HashSet`, simplemente sustituyendo el nombre de la clase `HashSet` por una de las otras. Ni `TreeSet`, ni `LinkedHashSet` admiten duplicados, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz `Set` (que es la interfaz que implementan).

**Ejemplos de utilización de los conjuntos `TreeSet` y `LinkedHashSet`.**

**Conjunto `TreeSet`**

```
TreeSet <Integer> t;  
  
t=new TreeSet<Integer>();  
  
t.add(new Integer(4));  
  
t.add(new Integer(3));  
  
t.add(new Integer(1));  
  
t.add(new Integer(99));  
  
for (Integer i:t) System.out.println(i);
```

*Salida por pantalla*

1 3 4 99

(el resultado sale ordenado por valor)

### **Conjunto LinkedHashSet**

```
LinkedHashSet <Integer> t;  
  
t=new LinkedHashSet<Integer>();  
  
t.add(new Integer(4));  
  
t.add(new Integer(3));  
  
t.add(new Integer(1));  
  
t.add(new Integer(99));  
  
for (Integer i:t) System.out.println(i);
```

*Salida por pantalla*

4 3 1 99

(los valores salen ordenados según el momento de inserción en el conjunto)

## Autoevaluación

**Un árbol cuyos nodos solo pueden tener un único nodo hijo, en realidad es una lista. ¿Verdadero o falso?**

Verdadero.

- 
- Falso.

Acertaste.

Incorrecto. Revisa el tema para entender en que has fallado.

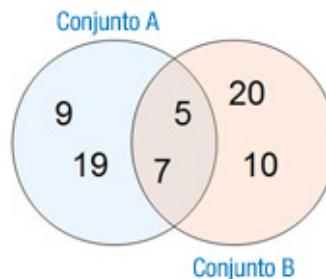
## Solución

1. Opción correcta
2. Incorrecto

### 3.3.- Conjuntos (IV).

¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle `for` y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se vieron en un apartado anterior, aquí simplemente vamos poner un ejemplo de su uso.

Partimos del siguiente ejemplo, en el que hay dos colecciones de diferente tipo, cada una con 4 números enteros:



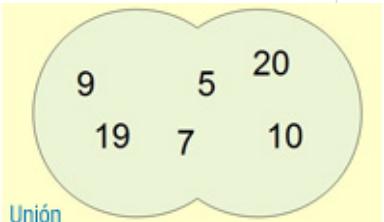
Salvador Romero Villegas ([CC BY-NC](#))

```
TreeSet<Integer> A= new TreeSet<Integer>();  
  
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7  
  
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();  
  
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio `Integer` sin tener que hacer nada, lo cual es una ventaja. Veamos las formas de combinar ambas colecciones:

#### Tipos de combinaciones.

Combinación.	Código.	Elementos finales del conjunto A.
--------------	---------	-----------------------------------

Combinación.	Código.	Elementos finales del conjunto A.
<b>Unión. Añadir todos los elementos del conjunto B en el conjunto A.</b>	<code>A.addAll(B)</code>	<p>Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.</p>  <p>Unión</p>
		<p>Ministerio de Educación y FP (<a href="#">CC BY-NC</a>)</p>
<b>Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.</b>	<code>A.removeAll(B)</code>	<p>Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.</p>  <p>Diferencia o sustracción</p>
		<p>Ministerio de Educación y FP (<a href="#">CC BY-NC</a>)</p>
<b>Intersección. Retiene los elementos comunes a ambos conjuntos.</b>	<code>A.retainAll(B)</code>	<p>Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.</p>  <p>Intersección</p>
		<p>Ministerio de Educación y FP (<a href="#">CC BY-NC</a>)</p>

Recuerda, estas operaciones son comunes a todas las colecciones.

## Para saber más

Puede que no recuerdes cómo era eso de los conjuntos, y dada la íntima relación de las colecciones con el álgebra de conjuntos, es recomendable que repases cómo era aquello, con el siguiente artículo de la Wikipedia.

[Álgebra de conjuntos.](#)

## Autoevaluación

Tienes un `HashSet` llamado `vocales` que contiene los elementos "a", "e", "i", "o", "u", y otro, llamado `vocales_fuertes` con los elementos "a", "e" y "o". ¿De qué forma podríamos sacar una lista con las denominadas vocales débiles (que son aquellas que no son fuertes)?

- `vocales.retainAll (vocales_fuertes);`
- `vocales.removeAll(vocales_fuertes);`
- No es posible hacer esto con `HashSet`, solo se puede hacer con `TreeSet` o `LinkedHashSet`.

No es correcto, no se trata de hacer una intersección.

Efectivamente, sería una diferencia.

Incorrecto. Como se ha dicho antes estas operaciones son comunes a todas las colecciones.

## Solución

1. Incorrecto
2. Opción correcta

3. Incorrecto

## 3.4.- Conjuntos (V).

Por defecto, los `TreeSet` ordenan sus elementos de forma ascendente, pero, ¿se podría cambiar el orden de ordenación? Los `TreeSet` tienen un conjunto de operaciones adicionales, además de las que incluye por el hecho de ser un conjunto, que permite entre otras cosas, cambiar la forma de ordenar los elementos. Esto es especialmente útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo por ejemplo). `TreeSet` es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.

Para indicar a un `TreeSet` cómo tiene que ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica `java.util.Comparator`, usada en general en algoritmos de ordenación, como veremos más adelante. Se trata de crear una clase que implemente dicha interfaz, así de fácil. Dicha interfaz requiere de un único método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo. Veamos un ejemplo general de cómo implementar un comparador para una hipotética clase "`Objeto`".

en:joestape89  
(CC BY-SA)

```
class ComparadorDeObjetos implements Comparator<Objeto> {  
  
    public int compare(Objeto o1, Objeto o2) { ... }  
  
}
```

La interfaz `Comparator` obliga a implementar un único método, es el método `compare`, el cual tiene dos parámetros: los dos elementos a comparar. Las reglas son sencillas, a la hora de personalizar dicho método:

- ✓ Si el primer objeto (`o1`) es menor que el segundo (`o2`), debe retornar un número entero negativo.
- ✓ Si el primer objeto (`o1`) es mayor que el segundo (`o2`), debe retornar un número entero positivo.
- ✓ Si ambos son iguales, debe retornar 0.

A veces, cuando el orden que deben tener los elementos es diferente al orden real (por ejemplo cuando ordenamos los números en orden inverso), la definición de antes puede ser un poco liosa, así que es recomendable en tales casos pensar de la siguiente forma:

- ✓ Si el primer objeto (`o1`) debe ir antes que el segundo objeto (`o2`), retornar entero negativo.
- ✓ Si el primer objeto (`o1`) debe ir después que el segundo objeto (`o2`), retornar entero positivo.
- ✓ Si ambos son iguales, debe retornar 0.

Una vez creado el comparador simplemente tenemos que pasarlo como parámetro en el momento de la creación al `TreeSet`, y los datos internamente mantendrán dicha ordenación:

```
TreeSet<Objeto> ts=new TreeSet<Objeto>(new ComparadorDeObjetos());
```

# Ejercicio resuelto

¿Fácil no? Pongámoslo en práctica. Imagínate que **Objeto** es una clase como la siguiente:

```
class Objeto {  
    public int a;  
    public int b;  
}
```

Imagina que ahora, al añadirlos en un **TreeSet**, estos se tienen que ordenar de forma que la suma de sus atributos (**a** y **b**) sea descendente, ¿cómo sería el comparador?

[Mostrar retroalimentación](#)

Una de las posibles soluciones a este problema podría ser la siguiente:

```
class ComparadorDeObjetos implements Comparador<Objeto> {  
  
    @Override  
  
    public int compare(Objeto o1, Objeto o2) {  
  
        int sumao1=o1.a+o1.b;  int sumao2=o2.a+o2.b;  
  
        if (sumao1<sumao2) return 1;  
  
        else if (sumao1>sumao2) return -1;  
  
        else return 0;  
  
    }  
}
```

## 4.- Listas (I).

### Caso práctico

Juan se queda pensando después de que Ana le preguntara si sabía los tipos de colecciones que había en Java. Obviamente no lo sabía, son muchos tipos, pero ya tenía una respuesta preparada:

—Bueno, sea lo que sea, siempre puedes utilizar una lista para almacenar lo que sea. Yo siempre las uso, pues te permiten almacenar cualquier tipo de objeto, extraer uno de las lista sin tener que recorrerla entera, buscar si hay o no un elemento en ella, de forma cómoda. Son para mi el mejor invento desde la rueda — dijo Juan.

—Ya, supongo, pero hay dos tipos de listas que me interesan, `LinkedList` y `ArrayList`, ¿cuál es mejor? ¿Cuál me conviene más? —respondió Ana.



Ministerio de Educación. ([CC BY-NC](#))

¿En qué se diferencia una lista de un conjunto? Las listas son elementos de programación un poco más avanzados que los conjuntos. Su ventaja es que amplían el conjunto de operaciones de las colecciones añadiendo operaciones extra, veamos algunas de ellas:

- ✓ Las listas si pueden almacenar duplicados, si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- ✓ Acceso posicional. Podemos acceder a un elemento indicando su posición en la lista.
- ✓ Búsqueda. Es posible buscar elementos en la lista y obtener su posición. En los conjuntos, al ser colecciones sin aportar nada nuevo, solo se podía comprobar si un conjunto contenía o no un elemento de la lista, retornando verdadero o falso. Las listas mejoran este aspecto.
- ✓ Extracción de sublistas. Es posible obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.



ITE. Ministerio de educación idITE=120475 ([CC BY-NC](#))

En Java, para las listas se dispone de una interfaz llamada `java.util.List`, y dos implementaciones (`java.util.LinkedList` y `java.util.ArrayList`), con diferencias significativas entre ellas. Los métodos de la interfaz List, que obviamente estarán en todas las implementaciones, y que permiten las operaciones anteriores son:

- ✓ `E get(int index)`. El método `get` permite obtener un elemento partiendo de su posición (`index`).
- ✓ `E set(int index, E element)`. El método `set` permite cambiar el elemento almacenado en una posición de la lista (`index`), por otro (`element`).
- ✓ `void add(int index, E element)`. Se añade otra versión del método `add`, en la cual se puede insertar un elemento (`element`) en la lista en una posición concreta (`index`), desplazando los existentes.

- ✓ `E remove(int index)`. Se añade otra versión del método `remove`, esta versión permite eliminar un elemento indicando su posición en la lista.
- ✓ `boolean addAll(int index, Collection<? extends E> c)`. Se añade otra versión del método `addAll`, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos.
- ✓ `int indexOf(Object o)`. El método `indexOf` permite conocer la posición (índice) de un elemento, si dicho elemento no está en la lista retornará -1.
- ✓ `int lastIndexOf(Object o)`. El método `lastIndexOf` nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista si puede almacenar duplicados).
- ✓ `List<E> subList(int from, int to)`. El método `subList` genera una sublista (una vista parcial de la lista) con los elementos comprendidos entre la posición inicial (incluida) y la posición final (no incluida).

Ten en cuenta que los elementos de una lista empiezan a numerarse por 0. Es decir, que el primer elemento de la lista es el 0. Ten en cuenta también que `List` es una interfaz genérica, por lo que `<E>` corresponde con el tipo base usado como parámetro genérico al crear la lista.

## Autoevaluación

Si `M` es una lista de números enteros, ¿sería correcto poner "`M.add(M.size(),3);`"?

- Sí.
- No.

Correcto. Inserta un elemento al final de la lista y es equivalente a poner `M.add(3)`.

Incorrecto. Piénsalo, ese código inserta un elemento al final de la lista.

## Solución

1. Opción correcta
2. Incorrecto

## 4.1.- Listas (II).

Y, ¿cómo se usan las listas? Pues para usar una lista haremos uso de sus implementaciones `LinkedList` y `ArrayList`. Veamos un ejemplo de su uso y después obtendrás respuesta a esta pregunta.

Supongo que intuirás como se usan, pero nunca viene mal un ejemplo sencillo, que nos aclare las ideas. El siguiente ejemplo muestra como usar un `LinkedList` pero valdría también para `ArrayList` (no olvides importar las clases `java.util.LinkedList` y `java.util.ArrayList` según sea necesario). En este ejemplo se usan los métodos de acceso posicional a la lista:



ITE. Francisco Javier Martínez Adrados.  
idITE=148928 ([CC BY-NC](#))

```
LinkedList<Integer> t=new LinkedList<Integer>(); // Declaración y creación del LinkedList de enteros.

t.add(1); // Añade un elemento al final de la lista.

t.add(3); // Añade otro elemento al final de la lista.

t.add(1,2); // Añade en la posición 1 el elemento 2.

t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.

t.remove(0); // Elimina el primer elementos de la lista.

for (Integer i: t) System.out.println("Elemento:" + i); // Muestra la lista.
```

En el ejemplo anterior, se realizan muchas operaciones, ¿cuál será el contenido de la lista al final? Pues será 2, 3 y 5. En el ejemplo cabe destacar el uso del bucle `for-each`, recuerda que se puede usar en cualquier colección.

Veamos otro ejemplo, esta vez con `ArrayList`, de cómo obtener la posición de un elemento en la lista:

```
ArrayList<Integer> al=new ArrayList<Integer>(); // Declaración y creación del ArrayList de enteros.

al.add(10); al.add(11); // Añadimos dos elementos a la lista.

al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazamos
```

En el ejemplo anterior, se emplea tanto el método `indexOf` para obtener la posición de un elemento, como el método `set` para reemplazar el valor en una posición, una combinación muy habitual. El ejemplo anterior generará un `ArrayList` que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

```
al.addAll(0, t.subList(1, t.size()));
```

Este ejemplo es especial porque usa sublistas. Se usa el método `size` para obtener el tamaño de la lista. Después el método `subList` para extraer una sublista de la lista (que incluía en origen los números 2, 3 y 5), desde la posición 1 hasta el final de la lista (lo cual dejaría fuera al primer elemento). Y por último, se usa el método `addAll` para añadir todos los elementos de la sublista al `ArrayList` anterior.

Debes saber que las operaciones aplicadas a una sublista repercuten sobre la lista original. Por ejemplo, si ejecutamos el método `clear` sobre una sublista, se borrarán todos los elementos de la sublista, pero también se borrarán dichos elementos de la lista original:

```
al.subList(0, 2).clear();
```

Lo mismo ocurre al añadir un elemento, se añade en la sublista y en la lista original.

## Debes conocer

Las listas enlazadas son un elemento muy recurrido y su funcionamiento interno es complejo. Te recomendamos el siguiente artículo de la wikipedia para profundizar un poco más en las listas enlazadas y los diferentes tipos que hay.

[Listas enlazadas.](#)

## Autoevaluación

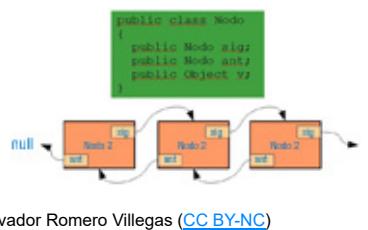
Completa con el número que falta.  
Dado el siguiente código:

```
LinkedList<Integer> t=new LinkedList<Integer>();  
  
t.add(t.size()+1); t.add(t.size()+1); Integer suma = t.get(0) + t.get(1);
```

El valor de la variable suma después de ejecutarlo es  .

## 4.2.- Listas (III).

¿Y en qué se diferencia un `LinkedList` de un `ArrayList`? Los `LinkedList` utilizan listas doblemente enlazadas, que son listas enlazadas (como se vio en un apartado anterior), pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.



Salvador Romero Villegas ([CC BY-NC](#))

No es el caso de los `ArrayList`. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redonda en una diferencia de rendimiento notable dependiendo del uso. Los `ArrayList` son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista). En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

¿Y esto que quiere decir? **Que si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (`LinkedList`), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (`ArrayList`).**

`LinkedList` tiene otras ventajas que nos puede llevar a su uso. Implementa las interfaces `java.util.Queue` y `java.util.Deque`. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Las colas, también conocidas como colas de prioridad, son una lista pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual. Se trata de que el que primero llega es el primero en ser atendido (FIFO). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (`add` y `offer`), sacar y eliminar el elemento más antiguo (`poll`), y examinar el elemento al principio de la lista sin eliminarlo (`peek`). Dichos métodos están disponibles en las listas enlazadas `LinkedList`:

- ✓ `boolean add(E e)` y `boolean offer(E e)`, retornarán true si se ha podido insertar el elemento al final de la `LinkedList`.
- ✓ `E poll()` retornará el primer elemento de la `LinkedList` y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará `null` si la lista está vacía.
- ✓ `E peek()` retornará el primer elemento de la `LinkedList` pero no lo eliminará, permite examinarlo. Retornará `null` si la lista está vacía.

Las pilas, mucho menos usadas, son todo lo contrario a las listas. Una pila es igual que una montaña de hojas en blanco, para añadir hojas nuevas se ponen encima del resto, y para retirar una se coge la primera que hay, encima de todas. En las pilas el último en llegar es el primero en ser atendido. Para ello se proveen de tres métodos: meter al principio de la pila (`push`), sacar y eliminar del principio de la pila (`pop`), y examinar el primer elemento de la pila (`peek`, igual que si se usara la lista como una cola).

Las pilas se usan menos y haremos menos hincapié en ellas. Simplemente ten en mente que, tanto las colas como las pilas, son una lista enlazada sobre la que se hacen operaciones especiales.

# Autoevaluación

Dada la siguiente lista, usada como si fuera una cola de prioridad, ¿cuál es la letra que se mostraría por la pantalla tras su ejecución?

```
LinkedList<String> tt=new LinkedList<String>();  
  
tt.offer("A"); tt.offer("B"); tt.offer("C");  
  
System.out.println(tt.poll());
```

- A.
- C.
- D.

Correcto. Efectivamente, el primero en entrar es el primero que sale.

Incorrecto. Revisa el funcionamiento de las colas.

No es correcto. El elemento D no está en la lista.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

## 4.3.- Listas (IV).

A la hora de usar las listas, hay que tener en cuenta un par de detalles, ¿sabes cuáles? Es sencillo, pero importante.

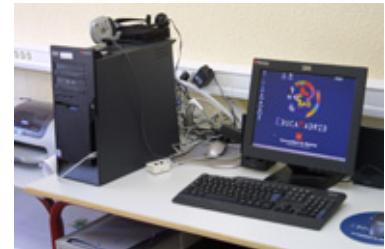
No es lo mismo usar las colecciones (listas y conjuntos) con objetos inmutables (`strings`, `Integer`, etc.) que con objetos mutables. Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos `add`, se pasan por copia (es decir, se realiza una copia de los mismos). En cambio los objetos mutables (como las clases que tú puedes crear), no se copian, y eso puede producir efectos no deseados.

Imaginate la siguiente clase, que contiene un número:

```
class Test
{
    public Integer num;

    Test (int num) { this.num=new Integer(num); }

}
```



ITE. Francisco Javier Martínez Adrados.  
idITE=148958 ([CC BY-NC](#))

La clase de antes es mutable, por lo que no se pasa por copia a la lista. Ahora imagina el siguiente código en el que se crea una lista que usa este tipo de objeto, y en el que se insertan dos objetos:

```
Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.

Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.

LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada para objetos tipo Test.

lista.add(p1); // Añadimos el primero objeto test.

lista.add(p2); // Añadimos el segundo objeto test.

for (Test p:lista) System.out.println(p.num); // Mostramos la lista de objetos.
```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12. Ahora bien, ¿qué pasa si modificamos el valor de uno de los números de los objetos test? ¿Qué se mostrará al ejecutar el siguiente código?

```
p1.num=44;
```

```
for (Test p:lista) System.out.println(p.num);
```

El resultado de ejecutar el código anterior es que se muestran los números 44 y 12. El número ha sido modificado y no hemos tenido que volver a insertar el elemento en la lista para que en la lista se cambie también. Esto es porque en la lista no se almacena una copia del objeto `Test`, sino un apuntador a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).

## Citas para pensar

"Controlar la complejidad es la esencia de la programación."

*Brian Kerniga*

## Autoevaluación

**Los elementos de un `ArrayList` de objetos `short` se copian al insertarse al ser objetos mutables. ¿Verdadero o falso?**

- Verdadero.
- Falso.

Lo siento, pero no has acertado. Los objetos `Short` son inmutables.

Acertaste. Los elementos se pasan por copia por ser inmutables, no mutables.

## Solución

1. Incorrecto
2. Opción correcta



## 5.- Conjuntos de pares clave/valor.

¿Cómo almacenarías los datos de un diccionario? Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los arrays asociativos. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En Java existe la interfaz `java.util.Map` que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: `java.util.HashMap`, `java.util.TreeMap` y `java.util.LinkedHashMap`. ¿Te suenan? Claro que sí. Cada una de ellas, respectivamente, tiene características similares a `HashSet`, `TreeSet` y `LinkedHashSet`, tanto en funcionamiento interno como en rendimiento.

Los mapas utilizan clases genéricas para dar extensibilidad y flexibilidad, y permiten definir un tipo base para la clave, y otro tipo diferente para el valor. Veamos un ejemplo de como crear un mapa, que es extensible a los otros dos tipos de mapas:

```
HashMap<String, Integer> t=new HashMap<String, Integer>();
```



Ministerio de Educación. ITE. idITE=111309  
(CC BY-NC)

El mapa anterior permite usar cadenas como llaves y almacenar de forma asociada a cada llave, un número entero. Veamos los métodos principales de la interfaz `Map`, disponibles en todas las implementaciones. En los ejemplos, `v` es el tipo base usado para el valor y `k` el tipo base usado para la llave:

### Métodos principales de los mapas.

Método.	Descripción.
<code>V put(K key, V value);</code>	Inserta un par de objetos llave ( <code>key</code> ) y valor ( <code>value</code> ) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará <code>null</code> .
<code>V get(Object key);</code>	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará <code>null</code> .
<code>V remove(Object key);</code>	Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o <code>null</code> , si la llave no existe.
<code>boolean containsKey(Object key);</code>	Retornará <code>true</code> si el mapa tiene almacenada la llave pasada por parámetro, <code>false</code> en cualquier otro caso.
<code>boolean containsValue(Object value);</code>	Retornará <code>true</code> si el mapa tiene almacenado el valor pasado por parámetro, <code>false</code> en cualquier otro caso.

Método.	Descripción.
int size();	Retornará el número de pares llave y valor almacenado en el mapa.
boolean isEmpty();	Retornará <code>true</code> si el mapa está vacío, <code>false</code> en cualquier otro caso.
void clear();	Vacía el mapa.

## Autoevaluación

**Completa el siguiente código para que al final se muestre el número 40 por pantalla:**

```
HashMap< String, [ ] > datos=new [ ] < String,String >();datos.  
[ ] ("A","44");System.out.println(Integer. [ ] (datos. [ ] (" [ ]  
"))-[ ]);
```

## 6.- Iteradores (I).

### Caso práctico

Juan se acerco a la mesa de Ana y le dijo:

—María me ha contado la tarea que te ha encomendado y he pensado que quizás te convendría usar mapas en algunos casos. Por ejemplo, para almacenar los datos del pedido asociados con una etiqueta: nombre, dirección, fecha, etc.

—La verdad es que pensaba almacenar los datos del pedido en una clase especial llamada Pedido. No tengo ni idea de que son los mapas -dijo Ana-, supongo que son como las listas, ¿tienen iteradores?

—Según me ha contado María, no necesitas hacer tanto, no es necesario crear una clase específica para los pedidos. Y respondiendo a tu pregunta, los mapas no tienen iteradores, pero hay una solución... te explico.



Ministerio de Educación. ITE. idITE=175787  
(CC BY-NC)

¿Qué son los iteradores realmente? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura. Los mapas, como no derivan de la interfaz Collection realmente, no tienen iteradores, pero como veremos, existe un truco interesante.

Los iteradores permiten recorrer las colecciones de dos formas: **bucles for-each** (existentes en Java a partir de la versión 1.5) **y a través de un bucle normal creando un iterador**. Como los bucles `for-each` ya los hemos visto antes (y ha quedado patente su simplicidad), nos vamos a centrar en el otro método, especialmente útil en versiones antiguas de Java. Ahora la pregunta es, ¿cómo se crea un iterador? Pues invocando el método "`iterator()`" de cualquier colección. Veamos un ejemplo (en el ejemplo `t` es una colección cualquiera):

```
Iterator<Integer> it=t.iterator();
```



Java. (usuario wikipedia que subió la imagen:  
[Machro](#)) (Dominio público)

Fíjate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "`<Integer>`" después de `Iterator`). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Si no se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornaría objetos tipo `Object` (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- ✓ `boolean hasNext()`. Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
- ✓ `E next()`. Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (`NoSuchElementException` para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
- ✓ `remove()`. Elimina de la colección el último elemento returned en la última invocación de `next` (no es necesario pasarselo por parámetro). Cuidado, si `next` no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (`while`) con la condición `hasNext()` nos permite hacerlo:

```
while (it.hasNext()) // Mientras que haya un siguiente elemento, seguiremos en el bucle.  
{  
  
    Integer t=it.next(); // Escogemos el siguiente elemento.  
  
    if (t%2==0) it.remove(); //Si es necesario, podemos eliminar el elemento extraído de la lista.  
  
}
```

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, solo los números impares.

## Reflexiona

Las listas permiten acceso posicional a través de los métodos `get` y `set`, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda de recorrer todos los elementos? ¿Un acceso posicional a través un bucle "`for (i=0;i<lista.size();i++)`" o un acceso secuencial usando un bucle "`while (iterador.hasNext())`"?

## 6.1.- Iteradores (II).

---

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto? En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el primer ejemplo se especifica el tipo de objeto del iterador, en el segundo ejemplo no, observa el uso de la conversión de tipos en la línea 6.

### Comparación de usos de los iteradores, con o sin conversión de tipos

#### Ejemplo indicando el tipo de objeto del iterador

```
ArrayList <Integer> lista=new ArrayList<Integer>();  
  
for (int i=0;i<10;i++) lista.add(i);  
  
Iterator<Integer> it=lista.iterator();  
  
while (it.hasNext()) {  
  
    Integer t=it.next();  
  
    if (t%2==0) it.remove();  
  
}
```

#### Ejemplo no indicando el tipo de objeto del iterador

```
ArrayList <Integer> lista=new ArrayList<Integer>();  
  
for (int i=0;i<10;i++) lista.add(i);  
  
Iterator it=lista.iterator();  
  
while (it.hasNext()) {  
  
    Integer t=(Integer)it.next();  
  
    if (t%2==0) it.remove();  
  
}
```

Un iterador es seguro porque está pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una incomoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método `entrySet` que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método `keySet` para generar un conjunto con las llaves existentes en el mapa. Veamos como sería para el segundo caso, el más sencillo:

```
HashMap<Integer, Integer> mapa=new HashMap<Integer, Integer>test();  
  
for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.  
  
for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet, contendrá las llaves  
{  
  
    Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si es necesario.  
  
}
```

Lo único que tienes que tener en cuenta es que el conjunto generado por `keySet` no tendrá obviamente el método `add` para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

## Recomendación

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método `remove` del iterador y no el de la colección. Si eliminas los elementos utilizando el método `remove` de la colección, mientras estás dentro de un bucle de iteración, o dentro de un bucle `for-each`, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

Los problemas son debidos a que el método `remove` del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interíormente información del orden de los elementos). Si usas el método `remove` de la colección, la información solo se elimina de un lugar, de la colección.

## Autoevaluación

¿Cuándo debemos invocar el método `remove()` de los iteradores?

- En cualquier momento.
- Despues de invocar el método `next()`.

- Despues de invocar el método `hasNext()`.
- No es conveniente usar este método para eliminar elementos, es mejor usar el de la colección.

Incorrecto. Deberías revisar el tema.

Efectivamente, has dado en el clavo.

No es correcto.

Incorrecto. Es relativo, si usas iteradores deberías usar este método, sino usas iteradores, no importa.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## Para saber más

A partir de Java 8, existen los conocidos **Java Stream Foreach**. Se trata de una nueva construcción que permite, entre otras cosas, recorrer colecciones utilizando una sintaxis muy cómoda y funcional.

En el siguiente enlace tienes una comparación entre los tradicionales bucles **Foreach** y **Java Stream Foreach**.

[Java Stream Foreach](#)

## 7.- Algoritmos (I).

### Caso práctico

**Ada** se acercó a preguntar a **Ana**. **Ada** era la jefa y **Ana** le tenía mucho respeto. **Ada** le preguntó cómo llevaba la tarea que le había encomendado **María**. Era una tarea importante, así que prestó mucha atención.

**Ana** le enseñó el código que estaba elaborando, le dijo que en un principio había pensado crear una clase llamada Pedido, para almacenar los datos del pedido, pero que Juan le recomendó usar mapas para almacenar los pares de valor y dato. Así que se decantó por usar mapas para ese caso. Le comentó también que para almacenar los artículos si había creado una pequeña clase llamada Artículo. Ada le dio el visto bueno:



Ministerio de Educación ([CC BY-NC](#))

—Pues Juan te ha recomendado de forma adecuada. Eso sí, sería recomendable que los artículos del pedido vayan ordenados por código de artículo —dijo Ada.

—¿Ordenar los artículos? Vaya, qué jaleo -respondió Ana.

—Arriba ese ánimo mujer, si has usado listas es muy fácil.

Ada explicó a Ana cómo mantener los artículos de un pedido ordenados por código de artículo. Inmediatamente después trató de dar implementación.

La palabra algoritmo seguro que te suena, pero, ¿a qué se refiere en el contexto de las colecciones y de otras estructuras de datos? Las colecciones, los arrays e incluso las cadenas, tienen un conjunto de operaciones típicas asociadas que son habituales. Algunas de estas operaciones ya las hemos visto antes, pero otras no. Veamos para qué nos pueden servir estas operaciones:

- ✓ Ordenar listas y arrays.
- ✓ Desordenar listas y arrays.
- ✓ Búsqueda binaria en listas y arrays.
- ✓ Conversión de arrays a listas y de listas a array.
- ✓ Partir cadenas y almacenar el resultado en un array.



ITE. Ministerio de educación  
idITE=110292 ([CC BY-NC](#))

Estos algoritmos están en su mayoría recogidos como métodos estáticos de las `clases java.util.Collections` y `java.util.Arrays`, salvo los referentes a cadenas obviamente.

Los algoritmos de ordenación ordenan los elementos en orden natural, siempre que Java sepa como ordenarlos. Como se explico en el apartado de conjuntos, cuando se desea que la

ordenación siga un orden diferente, o simplemente los elementos no son ordenables de forma natural, hay que facilitar un mecanismo para que se pueda producir la ordenación. Los tipos "ordenables" de forma natural son los enteros, las cadenas (orden alfabético) y las fechas, y por defecto su orden es ascendente.

La clase `Collections` y la clase `Arrays` facilitan el método `sort`, que permiten ordenar respectivamente listas y arrays. Los siguientes ejemplos ordenarían los números de forma ascendente (de menor a mayor):

## Ordenación natural en listas y arrays

### Ejemplo de ordenación de un array de números

```
Integer[] array={10,9,99,3,5};  
  
Arrays.sort(array);
```

### Ejemplo de ordenación de una lista con números

```
ArrayList<Integer> lista=new ArrayList<Integer>();  
  
lista.add(10); lista.add(9);lista.add(99);  
  
lista.add(3); lista.add(5);  
  
Collections.sort(lista);
```

## 7.1.- Algoritmos (II).

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. ¿Recuerdas la tarea que Ada pidió a Ana? Que los artículos del pedido aparecieran ordenados por código de artículo. Imagina que tienes los artículos almacenados en una lista llamada "**articulos**", y que cada artículo se almacena en la siguiente clase (fíjate que el código de artículo es una cadena y no un número):



ITE. Ministerio de educación. idITE=112221  
[\(CC BY-NC\)](#)

```
class Articulo {  
  
    public String codArticulo; // Código de artículo  
  
    public String descripcion; // Descripción del artículo.  
  
    public int cantidad; // Cantidad a proveer del artículo.  
  
}
```

La primera forma de ordenar consiste en crear una clase que implemente la interfaz **java.util.Comparator**, y en ende, el método **compare** definido en dicha interfaz. Esto se explicó en el apartado de conjuntos, al explicar el **TreeSet**, así que no vamos a profundizar en ello. No obstante, el comparador para ese caso podría ser así:

```
class comparadorArticulos implements Comparator<Articulo>  
{  
  
    @Override  
  
    public int compare( Articulo o1, Articulo o2) { return o1.codArticulo.compareTo(o2.codArticulo);  
  
}
```

Una vez creada esta clase, ordenar los elementos es muy sencillo, simplemente se pasa como segundo parámetro del método **sort** una instancia del comparador creado:

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma es quizás más sencilla cuando se trata de objetos cuya ordenación no existe de forma natural, pero requiere modificar la clase **Articulo**. Consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz **java.util.Comparable**.

**Todos los objetos que implementan la interfaz Comparable son "ordenables" y se puede invocar el método sort sin indicar un comparador para ordenarlos.** La interfaz comparable solo requiere implementar el método compareTo:

```
class Articulo implements Comparable<Articulo>{  
  
    public String codArticulo;  
  
    public String descripcion;  
  
    public int cantidad;  
  
    @Override  
  
    public int compareTo(Articulo o) { return codArticulo.compareTo(o.codArticulo); }  
  
}
```

Del ejemplo anterior se pueden denotar dos cosas importantes: que la interfaz Comparable es genérica y que para que funcione sin problemas es conveniente indicar el tipo base sobre el que se permite la comparación (en este caso, el objeto Articulo debe compararse consigo mismo), y que el método compareTo solo admite un parámetro, dado que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método compareTo es el mismo que el método compare de la interfaz Comparator: si la clase que se pasa por parámetro es igual al objeto, se tendría que retornar 0; si es menor o anterior, se debería retornar un número menor que cero; si es mayor o posterior, se debería retornar un número mayor que 0.

Ordenar ahora la lista de artículos es sencillo, fíjate que fácil: "collections.sort(articulos);"

## Autoevaluación

**Si tienes que ordenar los elementos de una lista de tres formas diferentes, ¿cuál de los métodos anteriores es más conveniente?**

- Usar comparadores, a través de la interfaz `java.util.Comparator`.
- Implementar la interfaz comparable en el objeto almacenado en la lista.

Efectivamente, creas tres comparadores, uno para cada forma de ordenar la lista.

No es la mejor forma, solo podrías ordenar los elementos de una forma.

# Solución

1. Opción correcta
2. Incorrecto

## 7.2.- Algoritmos (III).

¿Qué más ofrece las `clases java.util.Collections` y `java.util.Arrays` de Java? Una vez vista la ordenación, quizás lo más complicado, veamos algunas operaciones adicionales. En los ejemplos, la variable "array" es un array y la variable "lista" es una lista de cualquier tipo de elemento:

### Operaciones adicionales sobre listas y arrays.

Operación	Descripción	Ejemplos
<b>Desordenar una lista.</b>	Desordena una lista, este método no está disponible para arrays.	<code>Collections.shuffle(lista);</code>
<b>Rellenar una lista o array.</b>	Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array.	<code>Collections.fill(lista,elemento);</code> <code>Arrays.fill(array,elemento);</code>
<b>Búsqueda binaria.</b>	Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.	<code>Collections.binarySearch(lista,elemento);</code> <code>Arrays.binarySearch(array, elemento);</code>
<b>Convertir un array a lista.</b>	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornada (no es <code>ArrayList</code> ni <code>LinkedList</code> ), solo se especifica que retorna una lista que implementa la interfaz <code>java.util.List</code> .	<code>List lista=Arrays.asList(array);</code>  Si el tipo de dato almacenado en el array es conocido ( <code>Integer</code> por ejemplo), es conveniente especificar el tipo de objeto de la lista:  <code>List&lt;Integer&gt; lista = Arrays.asList(array);</code>

<b>Convertir una lista a array.</b>	Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase <code>Collections</code> , sino propio de la interfaz <code>Collection</code> . Es conveniente que sepas de su existencia.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista:
<b>Dar la vuelta.</b>	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	<pre>Integer[] array=new Integer[lista.size()]; lista.toArray(array)</pre>  <pre>Collections.reverse(lista);</pre>

Otra operación que no se ha visto hasta ahora es la dividir una cadena en partes. Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas. Es una operación sencilla, pero dado que es necesario conocer el funcionamiento de los arrays y de las expresiones regulares para su uso, no se ha podido ver hasta ahora. Para poder realizar esta operación, usaremos el método `split` de la clase `String`. El delimitador o separador es una expresión regular, único argumento del método `split`, y puede ser obviamente todo lo complejo que sea necesario:

```
String texto="Z,B,A,X,M,O,P,U";
String []partes=texto.split(",");
Arrays.sort(partes);
```



ITE. Ministerio de educación. idITE=109332  
(CC BY-NC)

En el ejemplo anterior la cadena `texto` contiene una serie de letras separadas por comas. La cadena se ha dividido con el método `split`, y se ha guardado cada carácter por separado en un array. Después se ha ordenado el array. ¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!

## Para saber más

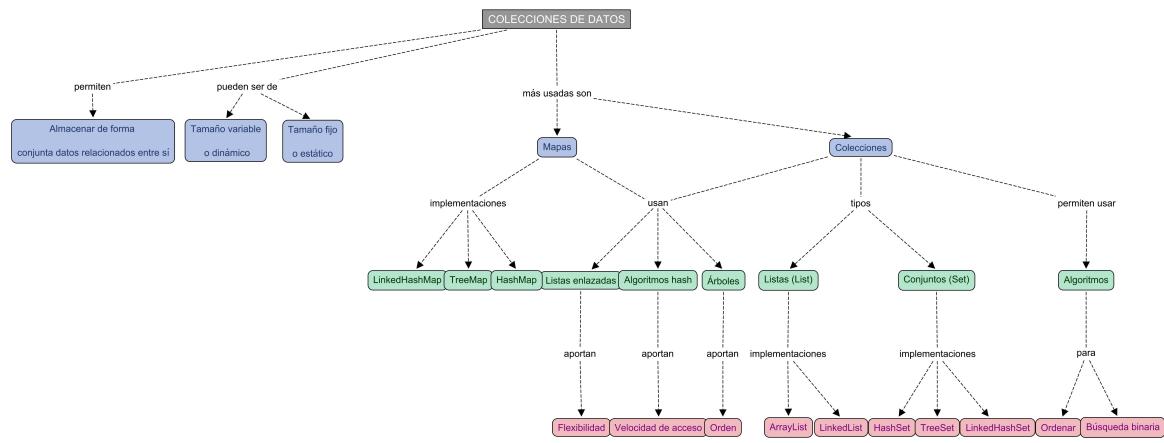
En el siguiente vídeo podrás ver en qué consiste la búsqueda binaria y cómo se aplica de forma sencilla:

<https://www.youtube.com/embed/-isTI614INQ>

[Resumen textual alternativo](#)

## 8.- Conclusiones.

Durante el desarrollo de esta unidad hemos trabajado con una de APIs más utilizadas de Java: el API Collections. Se trata de un conjunto de interfaces, clases y algoritmos para trabajar con una amplia gama de estructuras de datos dinámicas. Hemos aprendido a utilizar conjuntos, listas y árboles, algoritmos para su manipulación e iteradores para su recorrido de forma fácil y eficiente. Para su uso, es importante conocer el concepto de tipos genéricos.



Ministerio de Educación y FP (CC BY-NC)

En la siguiente Unidad nos centraremos en los mecanismos propuestos por el API Java para el intercambio de datos con memoria secundaria.

# Almacenando datos.

## Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

**Ada** está repasando los requisitos de la aplicación informática que están desarrollando para la clínica veterinaria.

En particular, ahora mismo se está centrándose en estudiar las necesidades respecto al almacenamiento de datos. **Ada** piensa que hay ciertas partes de la aplicación que no necesitan una base de datos para guardar los datos, y sería suficiente con emplear ficheros. Por ejemplo, para guardar datos de configuración de la aplicación.

Tras repasar, se reúne con **María** y **Juan** para planificar adecuadamente el tema de los ficheros que van a usar en la aplicación, ya que es un asunto muy importante, que no deben dejar apartado por más tiempo.

Precisamente **Antonio**, que cada vez está más entusiasmado con la idea de estudiar algún ciclo, de momento, está matriculado y cursando el módulo de Programación, y está repasando para el examen que tiene la semana que viene, uno de los temas que le "cae" es precisamente el de almacenamiento de información en ficheros.



[Ministerio de Educación y Formación Profesional.](#) (Dominio público)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**  
[Aviso Legal](#)

# 1.- Introducción.

Cuando desarrollas programas, en la mayoría de ellos los usuarios pueden pedirle a la aplicación que realice cosas y pueda suministrarte datos con los que se quiere hacer algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos, para proporcionar una respuesta a lo solicitado.

Además, normalmente interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros, que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

Por tanto, sabemos que el almacenamiento en variables o vectores (arrays) es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o cuando el programa termina. **Las computadoras utilizan ficheros para guardar los datos**, incluso después de que el programa termine su ejecución. Se suele denominar a los datos que se guardan en ficheros **datos persistentes**, porque existen, persisten más allá de la ejecución de la aplicación. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos cómo hacer con Java estas operaciones de crear, actualizar y procesar ficheros.

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como **Entrada/Salida (E/S)**.

Distinguimos dos tipos de E/S: la **E/S estándar** que se realiza con el terminal del usuario y la **E/S a través de ficheros**, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar del API de Java denominado `java.io` que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

El contenido de un archivo puede interpretarse como **campos** y **registros** (grupos de campos), dándole un significado al conjunto de bits que en realidad posee.

## Para saber más

A continuación puedes ampliar tus conocimientos sobre Entrada y Salida en general, en el mundo de la informática. Verás que es un basto tema lo que abarca.

[Entrada y Salida.](#)



Stephanie Booth (CC BY-NC)

## 1.1.- Excepciones.

Cuando se trabaja con archivos, es normal que pueda haber errores, por ejemplo: podríamos intentar leer un archivo que no existe, o podríamos intentar escribir en un archivo para el que no tenemos permisos de escritura. Para manejar todos estos errores debemos utilizar excepciones. Las dos excepciones más comunes al manejar archivos son:

- ✓ **FileNotFoundException**: Si no se puede encontrar el archivo.
- ✓ **IOException**: Si no se tienen permisos de lectura o escritura o si el archivo está dañado.



Luis Fernando Pienda Mahecha (CC BY)

Un esquema básico de uso de la captura y tratamiento de excepciones en un programa, podría ser este, importando el paquete **java.io.IOException**:

```
public static void main(String args[]) {  
    try {  
        // Se ejecuta algo que puede producir una excepción.  
        // catch (FileNotFoundException e) {  
        // manejo de una excepción por no encontrar un archivo  
  
        // catch (IOException e) {  
        // manejo de una excepción de entrada/salida  
  
        // catch (Exception e) {  
        // manejo de una excepción cualquiera  
  
        // finally {  
        // código a ejecutar haya o no excepción  
    }  
}
```

José Javier Bermúdez Hernández. ([CC BY-NC](#))

[Código de la estructura para gestionar excepciones.](#) (1.00 KB)

## Autoevaluación

Señala la opción correcta:

- Java no ofrece soporte para excepciones.
- Un campo y un archivo es lo mismo.
- Si se intenta abrir un archivo que no existe, entonces saltará una excepción.
- Ninguna es correcta.

Respuesta incorrecta, sí que las soporta.

Incorrecto, el contenido de un fichero está conformado por campos.

¡Exacto! Se disparará una excepción de tipo **FileNotFoundException**.

No has acertado, sí hay una correcta.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

## Para saber más

En el siguiente enlace hay un manual muy interesante de Java, en la web **w3schools**. Puedes consultar más información sobre las excepciones en Java así como de otros aspectos del lenguaje que te puedan interesar. La parte mas interesante de este portal es que explica de manera abreviada (en inglés) los contenidos con muchos ejemplos que además pueden ser ejecutados en línea. Además, contiene numerosos ejercicios en línea que te ayudarán a afianzar todos los contenidos. Otra parte muy interesante es que también tiene contenidos de otros lenguajes, como Javascript o PHP.

[Excepciones en Java.](#)

## 2.- Concepto de flujo.

### Caso práctico

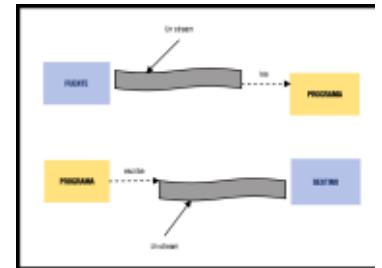
**Antonio** está estudiando un poco antes de irse a dormir. Se ha tomado un vaso de leche con cacao y está repasando el concepto de flujo. Entenderlo al principio, cuando lo estudió por primera vez, le costó un poco, pero ya lo entiende a la perfección y piensa que si le sale alguna pregunta en el examen de la semana que viene, sobre esto, seguro que la va a acertar.



Ministerio de Educación y FP ([CC BY-NC](#))

La clase `Stream` representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de entrada/salida (en adelante E/S), memoria, un conector TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet), etc.

Cualquier programa realizado en Java que necesite llevar a cabo una operación de entrada salida lo hará a través de un `stream`.



Ministerio de Educación y FP ([CC BY-NC](#))

Un flujo o stream es una abstracción de aquello que produzca o consuma información. Es una entidad lógica.

Las clases y métodos de E/S que necesitamos emplear son las mismas **independientemente del dispositivo con el que estemos actuando**, luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red liberando al programador de tener que saber con quién está interactuando.

La vinculación de un flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

En resumen, será el flujo el que tenga que comunicarse con el sistema operativo concreto y "entendérselas" con él. De esta manera, **no tenemos que cambiar absolutamente nada en nuestra aplicación**, que va a ser independiente tanto de los dispositivos físicos de

almacenamiento como del sistema operativo sobre el que se ejecuta. Esto es primordial en un lenguaje multiplataforma y tan altamente portable como Java.

## Autoevaluación

Señala la opción correcta:

- La clase `Stream` puede representar, al instanciarse, a un archivo.
- Si programamos en Java, hay que tener en cuenta el sistema operativo cuando tratemos con flujos, pues varía su tratamiento debido a la diferencia de plataformas.
- La clase `keyboard` es la clase a utilizar al leer flujos de teclado.
- La vinculación de un flujo al dispositivo físico la hace el hardware de la máquina.

¡Exacto!

¡No! Es indiferente.

¡Incorrecto! Es la clase `Stream`.

¡Incorrecto! Lo hace el sistema de E/S de Java.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## Para saber más

En la siguiente presentación puedes aprender más sobre sockets en Java.

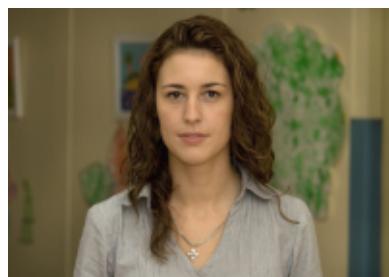
[Sockets en Java.](#)

[Resumen textual alternativo](#)

### 3.- Clases relativas a flujos.

## Caso práctico

Otro aspecto importante que **Ada** trata con **María y Juan**, acerca de los ficheros para la aplicación de la clínica, es el tipo de ficheros a usar. Es decir, deben estudiar si es conveniente utilizar ficheros para almacenar datos en ficheros de texto, o si deben utilizar ficheros binarios. María comenta -Quizás debemos usar los dos tipos de ficheros, dependerá de qué se vaya a guardar, -Juan le contesta -tienes razón María, pero debemos pensar entonces cómo va el programa a leer y a escribir la información, tendremos que utilizar las clases Java adecuadas según los ficheros que decidamos usar.



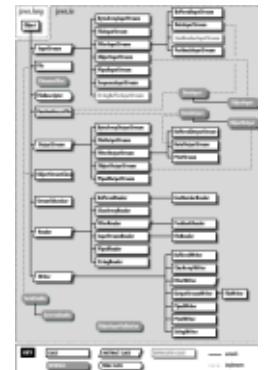
Ministerio de Educación y FP (CC BY-NC)

Existen dos tipos de flujos, flujos de bytes (byte streams) y flujos de caracteres (character streams).

- ✓ Los **flujos de caracteres** (16 bits) se usan para manipular datos legibles por humanos (por ejemplo un fichero de texto). Viene determinados por dos clases abstractas: **Reader** y **Writer**. Dichas clases manejan flujos de caracteres **Unicode**. De ellas derivan subclases concretas que implementan los métodos definidos. Destacados los métodos **read()** y **write()** que, en este caso, leen y escriben **caracteres** de datos respectivamente.
- ✓ Los **flujos de bytes** (8 bits) se usan para manipular datos binarios, legibles solo por la maquina (por ejemplo un fichero de programa). Su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son **InputStream** y **OutputStream**. Estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan **read()** y **write()** que leen y escriben bytes de datos respectivamente.

Las clases del paquete **java.io** se pueden ver en la ilustración. Destacamos las clases relativas a flujos:

- ✓ **BufferedInputStream**: permite leer datos a través de un flujo con un buffer intermedio.
- ✓ **BufferedOutputStream**: implementa los métodos para escribir en un flujo a través de un buffer.
- ✓ **FileInputStream**: permite leer bytes de un fichero.



O'Reilly & Associates (Todos los derechos reservados)

- ✓ `FileOutputStream`: permite escribir bytes en un fichero o descriptor.
- ✓ `StreamTokenizer`: esta clase recibe un flujo de entrada, lo analiza (parse) y divide en diversos pedazos (tokens), permitiendo leer uno en cada momento.
- ✓ `StringReader`: es un flujo de caracteres cuya fuente es una cadena de caracteres o string.
- ✓ `StringWriter`: es un flujo de caracteres cuya salida es un buffer de cadena de caracteres, que puede utilizarse para construir un string.

Destacar que hay clases que se **"montan"** sobre otros flujos para modificar la forma de trabajar con ellos. Por ejemplo, con `BufferedInputStream` podemos añadir un buffer a un flujo `FileInputStream`, de manera que se mejore la eficiencia de los accesos a los dispositivos en los que se almacena el fichero con el que conecta el flujo.

## Debes conocer

En el siguiente vídeo puedes clarificar algunos de estos conceptos.

<https://www.youtube.com/embed/-1C-wVnCd3c>

[Resumen textual alternativo](#)

## 3.1.- Ejemplo comentado de una clase con flujos.

Vamos a ver un ejemplo con una de las clases comentadas, en concreto, con `StreamTokenizer`.

La clase `StreamTokenizer` obtiene un flujo de entrada y lo divide en "tokens". El flujo tokenizer puede reconocer identificadores, números y otras cadenas.

El ejemplo que puedes descargar en el siguiente recurso, muestra cómo utilizar la clase `StreamTokenizer` para contar números y palabras de un fichero de texto. Se abre el flujo con ayuda de la clase `FileReader`, y puedes ver cómo se "monta" el flujo `StreamTokenizer` sobre el `FileReader`, es decir, que se construye el objeto `StreamTokenizer` con el flujo `FileReader` como argumento, y entonces se empieza a iterar sobre él.



[JD Hancock \(CC BY\)](#)

[Clase para leer palabras y números.](#)

El método `nextToken` devuelve un int que indica el tipo de token leído. Hay una serie de constantes definidas para determinar el tipo de token:

- ✓ `TT_WORD` indica que el token es una palabra.
- ✓ `TT_NUMBER` indica que el token es un número.
- ✓ `TT_EOL` indica que se ha leído el fin de línea.
- ✓ `TT_EOF` indica que se ha llegado al fin del flujo de entrada.

En el código de la clase, apreciamos que se iterará hasta llegar al fin del fichero. Para cada token, se mira su tipo, y según el tipo se incrementa el contador de palabras o de números.

### Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa.

Según el sistema operativo que utilicemos, habrá que utilizar un flujo u otro.  
¿Verdadero o Falso?

- Verdadero  Falso

**Verdadero**

El sistema operativo es indiferente, el flujo se encargará de los detalles subyacentes en la comunicación, por lo que el programador no tiene que preocuparse de esas cuestiones.

## 4.- Flujos.

### Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

**Ana y Antonio** salen de clase. Antonio ha quedado con una amiga y Ana va camino de casa pensando en lo que le explicaron en clase hace unos días. Como se quedó con dudas, también le consultó a María. En concreto, le asaltaban dudas sobre cómo leer y escribir datos por teclado en un programa, y también varias dudas sobre lectura y escritura de información en ficheros. **María** le solventó las dudas hablándole sobre el tema, pero aún así, tenía que probarlo tranquilamente en casa, haciéndose unos pequeños ejemplos, para comprobar toda la nueva información

aprendida.

-Antes de irte, -dice Antonio a Ana, -siéntate a hablar con nosotros un rato.

-Bueno, pero me voy a ir enseguida, -contesta Ana-.

Hemos visto qué es un flujo y que existe un árbol de clases amplio para su manejo. Ahora vamos a ver en primer lugar los **flujos predefinidos**, también conocidos como de entrada y salida, y después veremos los **flujos basados en bytes** y los **flujos basados en carácter**.



[David.nikonvscanon \(CC BY\)](#)

### Citas para pensar

**"Lo escuché y lo olvidé, lo vi y lo entendí, lo hice y lo aprendí".**

*Confucio.*

## 4.1.- Flujos predefinidos. Entrada y salida estándar.

Tradicionalmente, los usuarios del sistema operativo Unix, Linux y también MS-DOS, han utilizado un tipo de entrada/salida conocida comúnmente por entrada/salida estándar. El fichero de entrada estándar (`stdin`) es típicamente el teclado. El fichero de salida estándar (`stdout`) es típicamente la pantalla (o la ventana del terminal). El fichero de salida de error estándar (`stderr`) también se dirige normalmente a la pantalla, pero se implementa como otro fichero de forma que se pueda distinguir entre la salida normal y (si es necesario) los mensajes de error.



Ces-VLC (CC BY-NC)

Java tiene acceso a la entrada/salida estándar a través de la clase `System`. En concreto, los tres ficheros que se implementan son:

- ✓ `Stdin`. Es un objeto de tipo `InputStream`, y está definido en la clase `System` como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.
- ✓ `Stdout`. `System.out` implementa `stdout` como una instancia de la clase `PrintStream`. Se pueden utilizar los métodos `print()` y `println()` con cualquier tipo básico Java como argumento.
- ✓ `Stderr`. Es un objeto de tipo `PrintStream`. Es un flujo de salida definido en la clase `System` y representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.

Para la entrada, se usa el método `read` para leer de la entrada estándar:

- ✓ `int System.in.read();`
  - ↳ Lee el siguiente `byte` (`char`) de la entrada estándar.
- ✓ `int System.in.read(byte[] b);`
  - ↳ Leer un conjunto de bytes de la entrada estándar y lo almacena en el vector `b`.

Para la salida, se usa el método `print` para escribir en la salida estándar:

- ✓ `System.out.print(String);`
  - ↳ Muestra el texto en la consola.
- ✓ `System.out.println(String);`
  - ↳ Muestra el texto en la consola y seguidamente efectúa un salto de línea.

Normalmente, para **leer valores numéricos**, lo que se hace es tomar el valor de la entrada estándar en forma de cadena y entonces usar métodos que permiten transformar el texto a números (int, float, double, etc.) según se requiera.

### Funciones de conversión.

Método	Funcionamiento
<code>byte Byte.parseByte(String)</code>	Convierte una cadena en un número entero de un byte
<code>short Short.parseShort(String)</code>	Convierte una cadena en un número entero corto

Método	Funcionamiento
<code>int Integer.parseInt(String)</code>	Convierte una cadena en un número entero
<code>long Long.parseLong(String)</code>	Convierte una cadena en un número entero largo
<code>float Float.parseFloat(String)</code>	Convierte una cadena en un número real simple
<code>double Double.parseDouble(String)</code>	Convierte una cadena en un número real doble
<code>boolean Boolean.parseBoolean(String)</code>	Convierte una cadena en un valor lógico

## 4.2.- Flujos predefinidos. Entrada y salida estándar. Ejemplo.

Veamos un ejemplo en el que se lee por teclado hasta pulsar la tecla de retorno, en ese momento el programa acabará imprimiendo por la salida estándar la cadena leída.

Para ir construyendo la cadena con los caracteres leídos podríamos usar la clase **StringBuffer** o la **StringBuilder**. La clase **StringBuffer** permite almacenar cadenas que cambiarán en la ejecución del programa. **StringBuilder** es similar, pero no es síncrona. De este modo, para la mayoría de las aplicaciones, donde se ejecuta un solo hilo, supone una mejora de rendimiento sobre **StringBuffer**.

El proceso de lectura ha de estar en un bloque **try..catch**.

```
import java.io.IOException;
public class LeerSalida {
    public static void main(String[] args) {
        // Cada carácter que se introduce en la consola que no sea el carácter
        BufferedReader entrada = new BufferedReader(new InputStreamReader(
            System.in));
        // Por si ocurre una excepción ponemos el bloque try-catch
        try {
            // Recogemos la entrada de teclado no sea InputStream
            String linea = entrada.readLine();
            // Almacenar el carácter leído en la cadena str
            str.append(linea);
        } catch (IOException e) {
            System.out.println("Error");
        }
        // Imprimir la cadena que se ha sido recogido
        System.out.println("Cadena introducida: " + str);
    }
}
```

José Javier Bermúdez Hernández ([CC BY-NC](#))

[Código del proceso de lectura.](#)

## Autoevaluación

Señala la opción correcta:

- Read es una clase de System que permite leer caracteres.
- StringBuffer** permite leer y **StringBuilder** escribir en la salida estándar.
- La clase **keyboard** también permite leer flujos de teclado.
- Stderr por defecto dirige al monitor pero se puede direccional a otro dispositivo.

¡Incorrecto!

¡No es correcto!

¡No! Inténtalo de nuevo

¡Bien hecho! Esa es la respuesta correcta.

## Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

## Debes conocer

En este vídeo puedes ver un ejemplo sencillo y explicado sobre la gestión de las excepciones de I/O en Java.

<https://www.youtube.com/embed/2gWTVxe31g8>

[Resumen textual alternativo](#)

## 4.3.- Flujos basados en bytes.

---

Este tipo de flujos es el idóneo para el manejo de entradas y salidas de bytes, y su uso por tanto está orientado a la lectura y escritura de datos binarios.

Para el tratamiento de los flujos de bytes, Java tiene dos clases abstractas que son **InputStream** y **OutputStream**. Cada una de estas clases abstractas tiene varias subclases concretas, que controlan las diferencias entre los distintos dispositivos de E/S que se pueden utilizar.

```
class FileInputStream extends InputStream {  
  
    FileInputStream (String fichero) throws FileNotFoundException;  
  
    FileInputStream (File fichero) throws FileNotFoundException;  
  
    ... ... ...  
}  
  
class FileOutputStream extends OutputStream {  
  
    FileOutputStream (String fichero) throws FileNotFoundException;  
  
    FileOutputStream (File fichero) throws FileNotFoundException;  
  
    ... ... ...  
}
```

**OutputStream** y el **InputStream** y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Por ejemplo, podemos copiar el contenido de un fichero en otro:

```
void copia (String origen, String destino) throws IOException {  
    // Creamos los objetos de los dispositivos de origen y destino  
    // que queremos copiar.  
    InputStream entrada = new FileInputStream(origen);  
    OutputStream salida = new FileOutputStream(destino);  
    // Creamos una variable para leer el byte de lectura del origen  
    byte[] buffer = new byte[256];  
    while (true) {  
        // Leer el byte de lectura  
        int n = entrada.read(buffer);  
        // Si no queda nada para leer, salimos del bucle  
        if (n < 0) {  
            break;  
        }  
        // Escribir el byte de lectura leído al fichero destino  
        salida.write(buffer, 0, n);  
    }  
    // CERRAR LOS FICHIOS  
    // Entrada y salida  
    // Cierre el fichero  
    // Cierre el fichero  
    entrada.close();  
    salida.close();  
}
```

José Javier Bermúdez Hernández ([CC BY-NC](#))

[Código de copiar.](#)

## Recomendación

En los enlaces siguientes puedes ver la documentación oficial de Oracle sobre `FileInputStream` y sobre `FileOutputStream`.

[FileInputStream.](#)

[FileOutputStream.](#)

## 4.4.- Flujos basados en caracteres.

Las clases orientadas al flujo de bytes nos proporcionan la suficiente funcionalidad para realizar cualquier tipo de operación de entrada o salida, pero no pueden trabajar directamente con **caracteres Unicode**, los cuales están **representados por dos bytes**. Por eso, se consideró necesaria la creación de las clases orientadas al flujo de caracteres para ofrecernos el soporte necesario para el tratamiento de caracteres.

Para los flujos de caracteres, Java dispone de dos clases abstractas: **Reader** y **Writer**.

**Reader**, **Writer**, y todas sus subclases, reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Hay que recordar que **cada vez que se llama a un constructor se abre el flujo de datos y es necesario cerrarlo** cuando no lo necesitemos.

Existen muchos tipos de flujos dependiendo de la utilidad que le vayamos a dar a los datos que extraemos de los dispositivos.

**Un flujo puede ser envuelto por otro flujo para tratar el flujo de datos de forma cómoda.** Así, un **bufferWriter** nos permite manipular el flujo de datos como un buffer, pero si lo envolvemos en un **PrintWriter** lo podemos escribir con muchas más funcionalidades adicionales para diferentes tipos de datos.

En este ejemplo de código, se ve cómo podemos escribir la salida estándar a un fichero. Cuando se teclee la palabra "salir", se dejará de leer y entonces se saldrá del bucle de lectura.

Podemos ver cómo se usa **InputStreamReader** que es un puente de flujos de bytes a flujos de caracteres: lee bytes y los decodifica a caracteres. **BufferedReader** lee texto de un flujo de entrada de caracteres, permitiendo efectuar una lectura eficiente de caracteres, vectores y líneas.

Como vemos en el código, usamos **FileWriter** para flujos de caracteres, pues para datos binarios se utiliza **OutputStream**.

```
try{
    PrintWriter out = null;
    out = new PrintWriter(new FileWriter("Archivos\\linea.txt"));
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String s;
    while ((s = br.readLine()) != null) {
        if(s.equals("salir"))
            out.println(s);
    }
    out.close();
}
catch(IOException e)
{
    System.out.println(e.getMessage());
}
```

José Javier Bermúdez Hernández ([CC BY-NC](#))

### Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:



Macglee ([CC BY-NC-SA](#))

**Para flujos de caracteres es mejor usar las clases Reader y Writer en vez de InputStream y OutputStream.**

- Verdadero  Falso

**Verdadero**

Reader y Writer se aconseja para los ficheros binarios.

## Caso práctico

Trata de implementar en Netbeans el ejemplo mostrado en la imagen anterior. Recuerda que se trata de un algoritmo que lee de un fichero de texto línea a línea y las va mostrando por pantalla hasta encontrar la palabra "Salir".

Una mejora o ampliación podría ser que el mismo algoritmo vuelque el contenido del fichero origen a otro fichero de destino.

## 4.5.- Rutas de los ficheros.



Ministerio de Educación y FP ([CC BY-NC](#))

En los ejemplos que vemos en el tema estamos usando la ruta de los ficheros tal y como se usan en MS-DOS, o Windows, es decir, por ejemplo:

c:\datos\Programacion\fichero.txt

Cuando operamos con rutas de ficheros, el carácter separador entre directorios o carpetas suele cambiar dependiendo del sistema operativo en el que se esté ejecutando el programa.

Para evitar problemas en la ejecución de los programas cuando se ejecuten en uno u otro sistema operativo, y por tanto persiguiendo que nuestras aplicaciones sean lo más portables posibles, se recomienda usar en Java: `File.separator`.

Podríamos hacer una función que al pasarle una ruta nos devolviera la adecuada según el separador del sistema actual, del siguiente modo:

```
private String makeSeparator(String route) {
    String separator = "\\\\";
    if( // Si estamos en Windows
        (File.separator.equals("\\"))
        || route.contains("\\") ) // Si contiene una barra diagonal con la expresión
        // reemplaza cada barra por un carácter File.separator
        route = route.replace("\\", File.separator);
    else // Para el caso que Java utilice slashes para separar
        route = route.replace("/", File.separator);
    return route;
}
```

José Javier Bermúdez Hernández ([CC BY-NC](#))

[Código de separador de rutas.](#) (1 KB)

### Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación.

Cuando trabajamos con fichero en Java, no es necesario capturar las excepciones, el sistema se ocupa automáticamente de ellas. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

En efecto hay que tratarlas adecuadamente para evitar malos funcionamientos de la aplicación

## 5.- Trabajando con ficheros.

### Caso práctico

Juan le comenta a María -Tenemos que programar una copia de seguridad diaria de los datos del ficheros de texto plano que utiliza el programa para guardar la información. -Mientras María escucha a Juan, recuerda que para copias de seguridad, siempre ha comprobado que la mejor opción es utilizar ficheros secuenciales.



Ministerio de Educación y FP ([CC BY-NC](#))

¿Crees que es una buena opción la que piensa María o utilizarías otra en su lugar?

En este apartado vas a ver muchas cosas sobre los ficheros: cómo leer y escribir en ellos, aunque ya hemos visto algo sobre eso, hablaremos de las formas de acceso a los ficheros: secuencial o de manera aleatoria.

Siempre hemos de tener en cuenta que la manera de proceder con ficheros debe ser:



Ministerio de Educación y FP ([CC BY-NC](#))

- ✓ Abrir o bien **crear** si no existe el fichero.
- ✓ Hacer **las operaciones** que necesitemos.
- ✓ Cerrar el fichero, para no perder la información que se haya modificado o añadido.

También es muy importante el **control de las excepciones**, para evitar que se produzcan fallos en tiempo de ejecución. Si intentamos abrir sin más un fichero, sin comprobar si existe o no, y no existe, saltará una excepción.

### Para saber más

En el siguiente enlace a la wikipedia podrás ver la descripción de varias extensiones que pueden presentar los archivos.

[Extensión de un archivo.](#)

## 5.1.- Escritura y lectura de información en ficheros.

Acabamos de mencionar los pasos fundamentales para proceder con ficheros: abrir, operar, cerrar.

Además de esas consideraciones, debemos tener en cuenta también las clases Java a emplear, es decir, recuerda que hemos comentado que si vamos a tratar con ficheros de texto, es más eficiente emplear las clases de `Reader` `Writer`, frente a las clases de `InputStream` y `OutputStream` que están indicadas para flujos de bytes.



ITE (CC BY-NC)

Otra cosa a considerar, cuando se va a hacer uso de ficheros, es la forma de acceso al fichero que se va a utilizar, si va a ser de manera secuencial o bien aleatoria. En un fichero secuencial, **para acceder a un dato debemos recorrer todo el fichero desde el principio hasta llegar a su posición**. Sin embargo, en un fichero de acceso aleatorio podemos posicionarnos directamente en una posición del fichero, y ahí leer o escribir.

Aunque ya has visto un ejemplo que usa `BufferedReader`, insistimos aquí sobre la filosofía de estas clases, que usan la idea de un buffer.

La idea es que cuando una aplicación necesita leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le proporcione la información.

Un dispositivo cualquiera de memoria masiva, por muy rápido que sea, es mucho más lento que la CPU del ordenador.

Así que, es fundamental **reducir el número de accesos al fichero** a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia, el buffer, de modo que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo, ya que es una memoria mucho más rápida que cualquier otro dispositivo de memoria masiva.

Cualquier operación de Entrada/Salida a ficheros puede generar una `IOException`, es decir, un error de Entrada/Salida. Puede ser por ejemplo, que el fichero no exista, o que el dispositivo no funcione correctamente, o que nuestra aplicación no tenga permisos de lectura o escritura sobre el fichero en cuestión. Por eso, las sentencias que involucran operaciones sobre ficheros, deben ir en un bloque `try`.

### Autoevaluación

Señala si es verdadera o es falsa la siguiente afirmación:

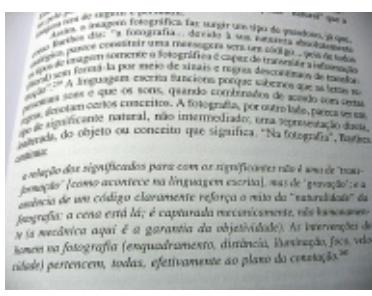
La idea de usar buffers con los ficheros es incrementar los accesos físicos a disco.  
¿Verdadero o falso?

- Verdadero  Falso

Falso

Se trata de disminuir esos accesos.

## 5.2.- Ficheros binarios y ficheros de texto (I).



[Entropier \(CC BY-NC\)](#)

Ya comentamos anteriormente que los ficheros se utilizan para guardar la información en un soporte: disco duro, disquetes, memorias usb, dvd, etc., y posteriormente poder recuperarla. También distinguimos dos tipos de ficheros: los de **texto** y los **binarios**.

En los **ficheros de texto** la información se guarda como caracteres. Esos caracteres están codificados en **Unicode**, o en **ASCII** u otras codificaciones de texto.

En la siguiente porción de código puedes ver cómo para un fichero existente, que en este caso es `texto.txt`, averiguamos la codificación que posee, usando el método `getEncoding()`

```
FileInputStream fichero;
try {
    // Creamos fichero para leer fijos de bytes "firma"
    fichero = new FileInputStream("C:\\\\texto.txt");
    // InputStreamReader sirve de puente de bytes a caracteres
    InputStreamReader lector = new InputStreamReader(fichero);
    // Vemos la codificación actual
    System.out.println(lector.getEncoding());
} catch (FileNotFoundException e) {
```

José Javier Bermúdez Hernández ([CC BY-NC](#))

### Código para mostrar codificación de fichero.

Para **archivos de texto**, se puede abrir el fichero para leer usando la clase **FileReader**. Esta clase nos proporciona métodos para **leer caracteres**. Cuando nos interese no leer carácter a carácter, sino **leer líneas completas**, podemos usar la clase **BufferedReader** a partir de **FileReader**. Lo podemos hacer de la siguiente forma:

```
File arch = new File ("C:\\\\fich.txt");

FileReader fr = new FileReader (arch);

BufferedReader br = new BufferedReader(fr);

...

String linea = br.readLine();
```

Para **escribir en archivos de texto** lo podríamos hacer, teniendo en cuenta:

```
FileWriter fich = null;

PrintWriter pw = null;

fich = new FileWriter("/fich2.txt");
```

```
pw = new PrintWriter(fichero);  
  
pw.println("Linea de texto");  
  
...  
  
Si el fichero al que queremos escribir existe y lo que queremos es añadir información, entonces pasaremos el segundo parámetro como true:
```

```
FileWriter("/fich2.txt",true);
```

## Para saber más

En el siguiente enlace a wikipedia puedes ver el código ASCII.

[Código Ascii.](#)

## 5.2.1.- Ficheros binarios y ficheros de texto (II).

Los **ficheros binarios** almacenan la información en bytes, codificada en binario, pudiendo ser de cualquier tipo: fotografías, números, letras, archivos ejecutables, etc.

Los archivos binarios guardan una representación de los datos en el fichero. O sea que, cuando se guarda texto no se guarda el texto en sí, sino que se guarda su representación en código UTF-8.

```
0080 FF DB FF E1 1D FE 45 78 69 66 00 00 49 49 2A 06  
0010 08 00 00 00 09 00 0F 01 02 00 06 00 00 00 7A 06  
0020 00 00 10 01 02 00 14 00 00 00 80 00 00 00 12 01  
0030 03 00 01 09 00 00 01 00 00 00 1A 01 05 00 01 09  
0040 00 00 00 09 00 00 01 00 00 00 01 01 00 00 00 09  
0050 00 00 2A 01 00 00 01 00 00 00 00 00 00 00 2A 01  
0060 02 00 14 00 00 00 00 00 00 00 00 00 12 02 03 00 01 08  
0070 00 00 01 00 00 00 00 49 87 04 00 01 00 00 00 00 C4 08  
0080 00 00 3A 06 00 00 43 61 6E GF 6E 00 41 61 6E GF  
0090 6E 20 50 6F 77 65 72 53 65 0F 74 20 41 36 39 08  
00A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00B0 01 00 00 00 00 00 00 00 01 00 00 00 00 00 32 30 30 34  
00C0 3A 30 36 3A 32 35 20 31 32 3A 33 39 34 32 35 09  
00D0 1F 00 9A 82 05 00 01 00 00 00 00 00 80 03 00 00 50 82  
00E0 05 00 01 00 00 00 00 03 00 00 00 00 00 00 07 00 04 06
```

[Paulinasca](#) (Dominio público)

Para **leer datos de un fichero binario**, Java proporciona la clase **FileInputStream**. Dicha clase trabaja con bytes que se leen desde el flujo asociado a un fichero. Aquí puedes ver un ejemplo comentado.

[Leer de fichero binario con buffer.](#) (3.00 KB)

Para **escribir datos a un fichero binario**, la clase nos permite usar un fichero para escritura de bytes en él, es la clase **FileOutputStream**. La filosofía es la misma que para la lectura de datos, pero ahora el flujo es en dirección contraria, desde la aplicación que hace de fuente de datos hasta el fichero, que los consume.

En la siguiente presentación puedes ver un esquema de cómo utilizar buffer para optimizar la lectura de teclado desde consola, por medio de las envolturas, podemos usar métodos como **readline()**, de la clase **BufferedReader**, que envuelve a un objeto de la clase **InputStreamReader**.

◀ 1 2 3 ▶

## Envolturas o Wrappers

### Leer desde consola

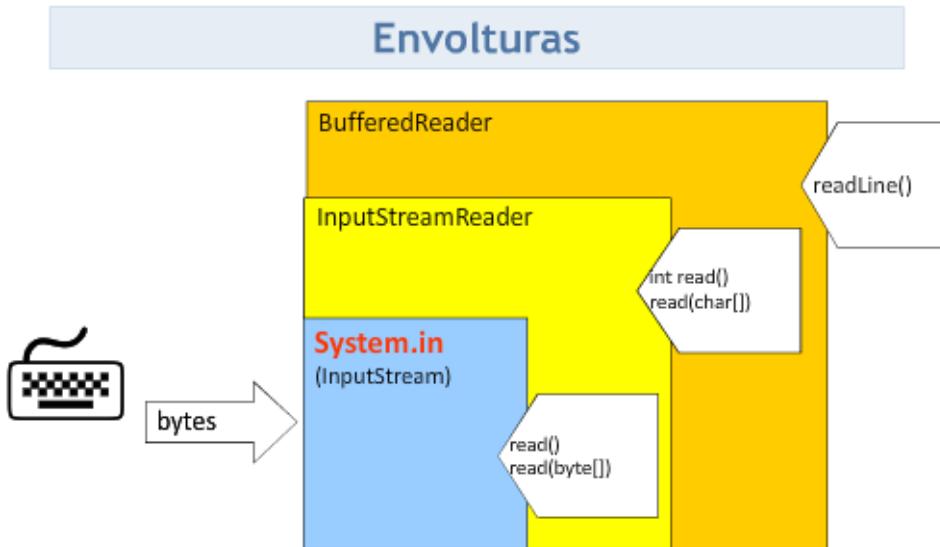
- **BufferedReader** buf = new BufferedReader(new InputStreamReader(System.in));

- buf es un flujo de caracteres que se enlaza a la consola a través de la clase System.in. Ésta se envuelve para pasar de byte a char.

- **InputStreamReader** ( InputStream inp): clase que convierte de byte a carácter.

- **BufferedReader** ( Reader input): clase que recibe un flujo de caracteres de entrada.

# Envolturas o Wrappers



# Envolturas o Wrappers

Todas las imágenes son propiedades del Ministerio de Educación y FP bajo licencia CC BY-NC.

[Resumen textual alternativo](#)

## Autoevaluación

**Señala si es verdadera o falsa la siguiente afirmación:**

Para leer datos desde un fichero codificados en binario empleamos la clase `FileOutputStream`. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

Se trata en efecto de `FileInputStream`.

## 5.3.- Modos de acceso. Registros.

En Java no se impone una estructura en un fichero, por lo que conceptos como el de registro que si existen en otros lenguajes, en principio no existen en los archivos que se crean con Java. Por tanto, los programadores deben estructurar los ficheros de modo que cumplan con los requerimientos de sus aplicaciones.

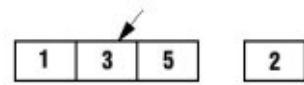
Así, el programador definirá su registro con el número de bytes que le interesen, moviéndose luego por el fichero teniendo en cuenta ese tamaño que ha definido.

Se dice que un fichero es de acceso directo o de organización directa cuando para acceder a un registro **n** cualquiera, no se tiene que pasar por los **n-1** registros anteriores. En caso contrario, estamos hablando de ficheros secuenciales.

Con Java se puede trabajar con **ficheros secuenciales** y con **ficheros de acceso aleatorio**.

En los **ficheros secuenciales**, la información se almacena de manera secuencial, de manera que para recuperarla se debe hacer en el mismo orden en que la información se ha introducido en el archivo. Si por ejemplo queremos leer el registro del fichero que ocupa la posición tres (en la ilustración sería el número 5), tendremos que abrir el fichero y leer los primeros tres registros, hasta que finalmente leamos el registro número tres.

Por el contrario, si se tratara de un **fichero de acceso aleatorio**, podríamos acceder directamente a la posición tres del fichero, o a la que nos interesa.



Ministerio de Educación y FP ([CC BY-NC](#))

### Autoevaluación

**Señala la opción correcta:**

- Java sólo admite el uso de ficheros aleatorios.
- Con los ficheros de acceso aleatorio se puede acceder a un registro determinado directamente.
- Los ficheros secuenciales se deben leer de tres en tres registros.
- Todas son falsas.

No es correcto, permite también los ficheros secuenciales.

¡Bien hecho! Esa es la respuesta correcta.

Incorrecto, no hay ninguna restricción de este tipo.

Respuesta incorrecta, repasa de nuevo los apuntes.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 5.4.- Acceso secuencial.



Ministerio de Educación y FP [\(CC BY-NC\)](#)

En el siguiente ejemplo vemos cómo se **escriben datos en un fichero secuencial**: el nombre y apellidos de una persona utilizando el método `writeUTF()` que proporciona `DataOutputStream`, seguido de su edad que la escribimos con el método `.writeInt()` de la misma clase. A continuación escribimos lo mismo para una segunda persona y de nuevo para una tercera. Después cerramos el fichero. Y ahora lo abrimos de nuevo para ir **leyendo de manera secuencial** los datos almacenados en el fichero, y escribiéndolos a consola.

[Escribir y leer.](#)

Fíjate al ver el código, que hemos tenido la precaución de ir escribiendo las cadenas de caracteres con el mismo tamaño, de manera que sepamos luego el tamaño del registro que tenemos que leer.

Por tanto para **buscar información en un fichero secuencial**, tendremos que abrir el fichero e ir leyendo registros hasta encontrar el registro que buscamos.

¿Y si queremos **eliminar un registro en un fichero secuencial**, qué hacemos? Esta operación es un problema, puesto que no podemos quitar el registro y reordenar el resto. Una opción, aunque costosa, sería crear un nuevo fichero. Recorremos el fichero original y vamos copiando registros en el nuevo hasta llegar al registro que queremos borrar. Ese no lo copiamos al nuevo, y seguimos copiando hasta el final, el resto de registros al nuevo fichero. De este modo, obtendríamos un nuevo fichero que sería el mismo que teníamos pero sin el registro que queríamos borrar. Por tanto, si se prevé que se va a borrar en el fichero, no es recomendable usar un fichero de este tipo, o sea, secuencial.

### Autoevaluación

**Señala si es verdadera o es falsa la siguiente afirmación:**

Para encontrar una información almacenada en la mitad de un fichero secuencial, podemos acceder directamente a esa posición sin pasar por los datos anteriores a esa información. ¿Verdadero o Falso?

- Verdadero  Falso

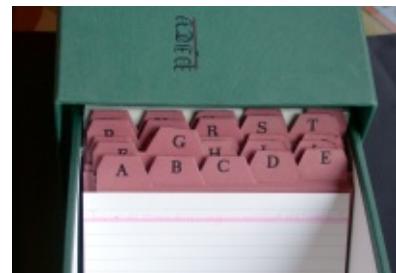
**Falso**

Se ha de recorrer el fichero desde el principio.

## 5.5.- Acceso aleatorio.

A veces no necesitamos leer un fichero de principio a fin, sino acceder al fichero como si fuera una base de datos, donde se accede a un registro concreto del fichero. Java proporciona la clase `RandomAccessFile` para este tipo de entrada/salida.

La clase `RandomAccessFile` permite utilizar un fichero de **acceso aleatorio** en el que el programador define el formato de los registros.



Ministerio de Educación y FP ([CC BY-NC](#))

```
RandomAccessFile objFile = new RandomAccessFile( ruta, modo );
```

Donde ruta es la dirección física en el sistema de archivos y modo puede ser:

- ✓ "r" para sólo lectura.
- ✓ "rw" para lectura y escritura.

La clase `RandomAccessFile` implementa los interfaces `DataInput` y `DataOutput`. Para abrir un archivo en modo lectura haríamos:

```
RandomAccessFile in = new RandomAccessFile("input.dat", "r");
```

Para abrirlo en modo lectura y escritura:

```
RandomAccessFile inOut = new RandomAccessFile("input.dat", "rw");
```

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases diferentes.

Hay que especificar el modo de acceso al construir un objeto de esta clase: sólo lectura o lectura/escritura.

Dispone de métodos específicos de desplazamiento como `seek` y `skipBytes` para poder moverse de un registro a otro del fichero, o posicionarse directamente en una posición concreta del fichero.

No está basada en el concepto de flujos o streams.

En el siguiente enlace tienes información general sobre el uso de ficheros de acceso secuencial en Java. Introduce los métodos más utilizados para el uso de este tipo de ficheros y además contiene tres ejemplos muy útiles y didácticos.

[Uso de ficheros de acceso aleatorio](#)

# Autoevaluación

Indica si es verdadera o es falsa la siguiente afirmación:

Para decirle el modo de lectura y escritura a un objeto `RandomAccessFile` debemos pasar como parámetro "rw". ¿Verdadero o Falso?

- Verdadero  Falso

**Verdadero**

Esos son los parámetros para el modo lectura y escritura.

## 6.- Aplicaciones del almacenamiento de información en ficheros.

### Caso práctico

**Antonio** ha quedado con **Ana** para estudiar sobre el tema de ficheros. De camino a la biblioteca, Ana le pregunta a Antonio -¿Crees que los ficheros se utilizan realmente, o ya están desfasados y sólo se utilizan las bases de datos? -Antonio tras pensarlo un momento le dice a Ana -Yo creo que sí, piensa en el mp3 que usas muchas veces, la música va grabada en ese tipo de ficheros.



Ministerio de Educación y FP (CC BY-NC)

¿Has pensado la **diversidad de ficheros** que existe, según la información que se guarda?

Las fotos que haces con tu cámara digital, o con el móvil, se guardan en ficheros. Así, existe una gran cantidad de ficheros de imagen, según el método que usen para guardar la información. Por ejemplo, tenemos los ficheros de extensión: .jpg, .tiff, gif, .bmp, etc.

La música que oyes en tu mp3 o en el reproductor de mp3 de tu coche, está almacenada en ficheros que almacenan la información en formato mp3.

Los sistemas operativos, como Linux, Windows, etc., están constituidos por un montón de instrucciones e información que se guarda en ficheros.

El propio código fuente de los lenguajes de programación, como Java, C, etc., se guarda en ficheros de texto plano la mayoría de veces.

También se guarda en ficheros las películas en formato .avi, .mp4, etc.

Y por supuesto, se usan mucho actualmente los ficheros XML, que al fin y al cabo son ficheros de texto plano, pero que siguen una estructura determinada. XML se emplea mucho para el intercambio de información estructurada entre diferentes plataformas.

```
<?xml version="1.0"?>
<quiz>
<question>
Who was the forty-second
president of the U.S.A.?
</question>
<answer>
William Jefferson Clinton
</answer>
<!-- Note: We need to add
more questions later.-->
</quiz>
```

XML

Dreftymac (CC BY-SA)

# Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:

Un fichero .bmp guarda información de música codificada. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

Se trata de una imagen, mp3 sería un ejemplo de fichero para guardar música.

## 7.- Utilización de los sistemas de ficheros.

### Caso práctico

**Ana** está estudiando en la biblioteca, junto a **Antonio**. Está repasando lo que le explicaron en clase sobre las operaciones relativas a ficheros en Java. En concreto, está mirando lo relativo a crear carpetas o directorios, listar directorios, borrarlos, operar en definitiva con ellos. Va a repasar ahora en la biblioteca, para tener claros los conceptos y cuando llegue de vuelta a casa, probar a compilar algunos ejemplos que a ella misma se le ocurran.



Ministerio de Educación y FP (CC BY-NC)

Has visto en los apartados anteriores cómo operar en ficheros: abrirlos, cerrarlos, escribir en ellos, etc.

Lo que no hemos visto es lo relativo a crear y borrar directorios, poder filtrar archivos, es decir, buscar sólo aquellos que tengan determinada característica, por ejemplo, que su extensión sea: `.txt`.

Ahora veremos cómo hacer estas cosas, y también como borrar ficheros, y crearlos, aunque crearlos ya lo hemos visto en algunos ejemplos anteriores.



Tim Morgan (CC BY)

### Para saber más

Accediendo a este enlace, tendrás una visión detallada sobre la organización de ficheros.

[Organización de Ficheros y Métodos de Enlace](#)

## 7.1.- Clase File.

La clase `File` proporciona una representación abstracta de ficheros y directorios.

Esta clase, permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase `File` representan nombres de archivo, no los archivos en sí mismos.



Vicente Villamón (CC BY-SA)

El archivo correspondiente a un nombre dado podría ser que no existiera, por ello, habrá que controlar las posibles excepciones.

Al trabajar con `File`, las rutas pueden ser:

- ✓ Relativas al directorio actual.
- ✓ Absolutas si la ruta que le pasamos como parámetro empieza por
  - ↳ La barra "/" en Unix, Linux.
  - ↳ Letra de unidad (C:, D:, etc.) en Windows.
  - ↳ UNC(universal naming convention) en windows, como por ejemplo:

```
File miFile=new File("\\\\mimaquina\\\\download\\\\prueba.txt");
```

A través del objeto `File`, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Dado un objeto `file`, podemos hacer las siguientes operaciones con él:

- ✓ **Renombrar** el archivo, con el método `renameTo()`. El objeto `File` dejará de referirse al archivo renombrado, ya que el `String` con el nombre del archivo en el objeto `File` no cambia.
- ✓ **Borrar** el archivo, con el método `delete()`. También, con `deleteOnExit()` se borra cuando finaliza la ejecución de la máquina virtual Java.
- ✓ **Crear** un nuevo fichero con un nombre único. El método estático `createTempFile()` crea un fichero temporal y devuelve un objeto `File` que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándonos tener un nombre de archivo no repetido.
- ✓ **Establecer** la fecha y la hora de modificación del archivo con `setLastModified()`. Por ejemplo, se podría hacer: `new File("prueba.txt").setLastModified(new Date().getTime());` para establecerle la fecha actual al fichero que se le pasa como parámetro, en este caso `prueba.txt`.
- ✓ **Crear** un directorio con el método `mkdir()`. También existe `mkdirs()`, que crea los directorios superiores si no existen.
- ✓ **Listar** el contenido de un directorio. Los métodos `list()` y `listFiles()` listan el contenido de un directorio. `list()` devuelve un vector de `String` con los nombres de los archivos, `listFiles()` devuelve un vector de objetos `File`.
- ✓ **Listar** los nombres de archivo de la raíz del sistema de archivos, mediante el método estático `listRoots()`.

# Autoevaluación

Indica si es verdadera o falsa la siguiente afirmación:

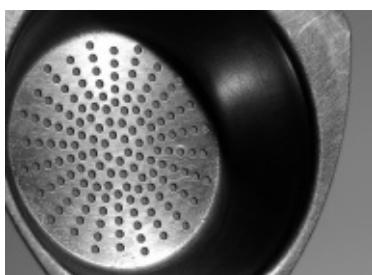
Un objeto de la clase File representa un fichero en sí mismo. ¿Verdadero o falso?

- Verdadero  Falso

**Falso**

En efecto, representa un nombre de archivo y no el archivo en sí mismo.

## 7.2.- Interface FilenameFilter.



Pau Bou (CC BY-NC-SA)

En ocasiones nos interesa ver la lista de los archivos que encajan con un determinado criterio.

Así, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que indiquemos, etc.

El interface `FilenameFilter` se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. Una clase que lo implemente debe definir e implementar el método:

```
boolean accept(File dir, String nombre)
```

Este método devolverá verdadero (`true`), en el caso de que el fichero cuyo nombre se indica en el parámetro `nombre` aparezca en la lista de los ficheros del directorio indicado por el parámetro `dir`.

En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta `c:\datos` que tengan la extensión `.odt`. Usamos `try` y `catch` para capturar las posibles excepciones, como que no exista dicha carpeta.

```
public class Filtro implements FilenameFilter {
    String extension;
    Filtro(String extension) {
        this.extension=extension;
    }
    @Override
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }
}

public static void main(String[] args) {
    try {
        File directorio = new File("c:\\datos\\");
        String[] listaArchivos = directorio.list(new Filtro(".odt"));
        int numArchivos = listaArchivos.length;

        if (numArchivos < 1)
            System.out.println("No hay archivos que listar");
        else
        {
            for (String listaArchivo : listaArchivos) {
                System.out.println(listaArchivo);
            }
        }
    } catch (Exception ex) {
        System.out.println("Error al buscar en la ruta indicada");
    }
}
```

José Javier Bermúdez Hernández. (CC BY-NC)

[Filtrar ficheros.](#) (2.00 KB)

## Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa:

Una clase que implemente `FileNameFilter` puede o no implementar el método `accept`. ¿Verdadero o Falso?

- Verdadero  Falso

**Falso**

En efecto, se debe implementar siempre.

## 7.3.- Creación y eliminación de ficheros y directorios.

---

Podemos **crear un fichero** del siguiente modo:

- ✓ Creamos el objeto que encapsula el fichero, por ejemplo, suponiendo que vamos a crear un fichero llamado miFichero.txt, en la carpeta C:\\prueba, haríamos:

```
File fichero = new File("c:\\\\prueba\\\\miFichero.txt");
```

- ✓ A partir del objeto **File** creamos el fichero físicamente, con la siguiente instrucción, que devuelve un boolean con valor true si se creó correctamente, o false si no se pudo crear:

```
fichero.createNewFile()
```

Para **borrar un fichero**, podemos usar la clase **File**, comprobando previamente si existe, del siguiente modo:

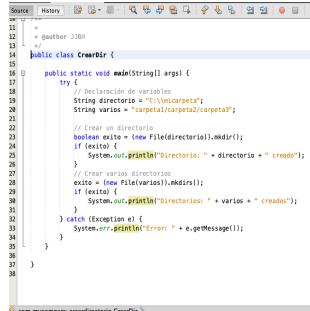
- ✓ Fijamos el nombre de la carpeta y del fichero con:

```
File fichero = new File("C:\\\\prueba", "agenda.txt");
```

- ✓ Comprobamos si existe el fichero con **exists()** y si es así lo borramos con:

```
fichero.delete();
```

Para **crear directorios**, podríamos hacer:



```
1 package com.jibhi;
2
3 /**
4 * Author: JIBHI
5 */
6
7 public class CrearDir {
8
9     public static void main(String[] args) {
10         try {
11             // Declaración de variables
12             String directorio = "C:\\\\Unarpeta";
13             String varios = "carpeta1\\carpeta2\\carpeta3";
14
15             // Crear un directorio
16             directorio = new File(directorio).mkdir();
17             if (textos)
18                 System.out.println("Directorio: " + directorio + " creado");
19
20             // Crear varios directorios
21             varios = new File(varios).mkdirs();
22             if (textos)
23                 System.out.println("Directorios: " + varios + " creados");
24
25         } catch (Exception e) {
26             System.err.println("Error: " + e.getMessage());
27         }
28     }
29
30 }
31
32
33
34
35
36
37
38
```

José Javier Bermúdez Hernández ([CC BY-NC](#))

### Crear directorios.

Para **borrar un directorio** con Java tenemos que borrar cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros.

Se puede listar el contenido del directorio con:

```
File[] ficheros = directorio.listFiles();
```

y entonces poder ir borrando. Si el elemento es un directorio, lo sabemos mediante el método **isDirectory**,

## 8.- Almacenamiento de objetos en ficheros. Persistencia. Serialización.

### Caso práctico

Ana ya tiene los conocimientos suficientes para llevar a cabo la tarea de procesar el fichero de pedidos y poder cargarlo en una estructura de datos en memoria. Con lo aprendido en esta unidad consigue la pieza que necesita para abordar la tarea. Así:

1. Para el tratamiento del fichero podrá utilizar las clases estudiadas que permiten procesar y leer un fichero de texto línea a línea.
2. Para la validación de cada línea leída decidió utilizar expresiones regulares, que ya ha probado y validado.
3. Por último, decisión que también tenía tomada, decidió utilizar varias estructuras para almacenar los pedidos: un mapa de pedidos y una lista de artículo ordenada por código de artículo.

En el siguiente enlace puedes acceder al código completo comentado desarrollada por Ana.

[Proyecto Procesar Pedidos](#)

¡Échale un vistazo al código y tratar de implementarlo y ejecutarlo!

### Caso práctico

Para la aplicación de la clínica veterinaria **María** le propone a **Juan** emplear un fichero para guardar los datos de los clientes de la clínica. -Como vamos a guardar datos de la clase Cliente, tendremos que serializar los datos.



Ministerio de Educación y FP (CC BY-NC)

¿Qué es la **serialización**? Es un proceso por el que **un objeto se convierte en una secuencia de bytes** con la que más tarde se podrá reconstruir el valor de sus variables. Esto permite guardar un objeto en un archivo.

Para serializar un objeto:

- ✓ éste debe **implementar el interface java.io.Serializable**. Este interface no tiene métodos, sólo se usa para informar a la JVM (Java Virtual Machine) que un objeto va a ser serializado.
- ✓ Todos los objetos incluidos en él tienen que implementar el interfaz Serializable.



[Ballistik Coffee Boy \(CC BY\)](#)

Todos los **tipos primitivos en Java son serializables** por defecto. (Al igual que los arrays y otros muchos tipos estándar).

Para leer y escribir objetos serializables a un stream se utilizan las clases java: **ObjectInputStream** y **ObjectOutputStream**.

En el siguiente ejemplo se puede ver cómo leer un objeto serializado que se guardó antes. En este caso, se trata de un String serializado:

```
FileInputStream fich = new FileInputStream("str.out");  
  
ObjectInputStream os = new ObjectInputStream(fich);  
  
Object o = os.readObject();
```

Así vemos que **readObject** lee un objeto desde el flujo de entrada **fich**. Cuando se leen objetos desde un flujo, se debe tener en cuenta qué tipo de objetos se esperan en el flujo, y se han de leer en el mismo orden en que se guardaron.

## Autoevaluación

**Indica si es verdadera o falsa la siguiente afirmación:**

Para serializar un fichero basta con implementar el interface **Serializable**.  
¿Verdadero o falso?

Verdadero  Falso

**Falso**

Debemos asegurarnos también de que todos los objetos incluidos en él implementen el interface Serializable.

## Para saber más

En el siguiente enlace a puedes ver un poco más sobre serialización.

[Serialización en Java.](#)

## 8.1.- Serialización: utilidad.



José Javier Bermúdez Hernández. ([CC BY-NC](#))

La serialización en Java se desarrolló para utilizarse con RMI. RMI necesitaba un modo de convertir los parámetros necesarios a enviar a un objeto en una máquina remota, y también para devolver valores desde ella, en forma de flujos de bytes. Para datos primitivos es fácil, pero para objetos más complejos no tanto, y ese mecanismo es precisamente lo que proporciona la serialización.

El método `writeObject` se utiliza para guardar un objeto a través de un flujo de salida. El objeto pasado a `writeObject` debe implementar el interfaz `Serializable`.

```
FileOutputStream fisal = new FileOutputStream("cadenas.out");  
  
ObjectOutputStream oos = new ObjectOutputStream(fisal);  
  
oos.writeObject();
```

La serialización de objetos se emplea también en la arquitectura de componentes software JavaBean. Las clases bean se cargan en herramientas de construcción de software visual, como NetBeans. Con la paleta de diseño se puede personalizar el bean asignando fuentes, tamaños, texto y otras propiedades.

Una vez que se ha personalizado el bean, para guardarlo, se emplea la serialización: se almacena el objeto con el valor de sus campos en un fichero con extensión .ser, que suele emplazarse dentro de un fichero .jar.

### Para saber más

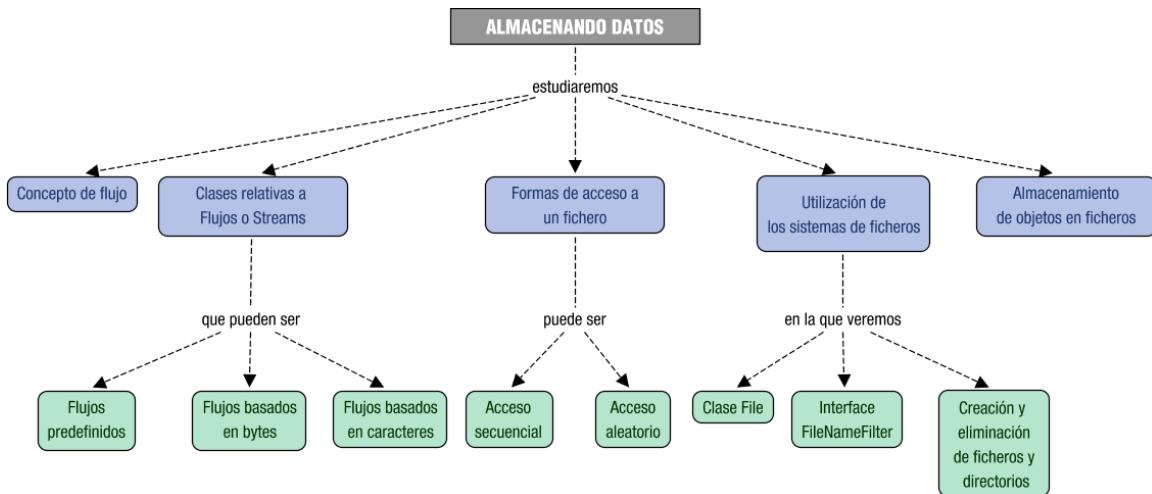
En este enlace a puedes ver un vídeo en el que se crea una aplicación sobre serialización. No está hecha con NetBeans, sino con Eclipse, pero eso no presenta ningún inconveniente.

<https://www.youtube.com/embed/4eU6WMOVmh4>

[Resumen textual alternativo](#)

## 9.- Conclusiones

La persistencia de datos en memoria secundaria es un requisito fundamental para las aplicaciones software: es la única forma de que los datos no se pierden cuando la aplicación finaliza la ejecución. En esta unidad hemos trabajado con todo tipo de ficheros en memoria secundaria para almacenar tanto datos binarios como caracteres. Además, hemos conocido el concepto de flujo o stream. Como hemos podido comprobar, a través del paquete `java.io`, el API Java proporciona clases e interfaces para que el almacenamiento y recuperación de datos hacia/desde ficheros sea una tarea sencilla.



Ministerio de Educación y FP ([CC BY-NC](#))

Un paso más en la persistencia de datos será el uso de base de datos, algo que dejaremos para la última unidad del curso. En la siguiente unidad nos centraremos en la construcción de interfaces gráficas de usuario.