1.- Introducción.

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.

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.

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.

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.

Paradigma	Relación	Definición
Programación Orientada a Objetos.		1. Maneja funciones y procedimientos que definen las acciones a realizar.
Programación Estructurada.		2. Representa las entidades del mundo real mediante componentes de la aplicación.

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.

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 q**ue 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.

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:

- **Comprensión**. 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.

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.

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 Stoustrup 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++.

Autoevaluación

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

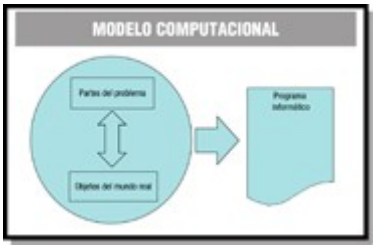
Lenguaje de programación	Relación	Tiene la característica de que …
Java		1. Fue el primer lenguaje que introdujo el concepto de clase.
SmallTalk		2. Introdujo el concepto del Modelo-Vista-Controlador.
Simula		3. Produce un bytecode para ser interpretado por la máquina virtual.
C++		4. Introduce el concepto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

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

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**.

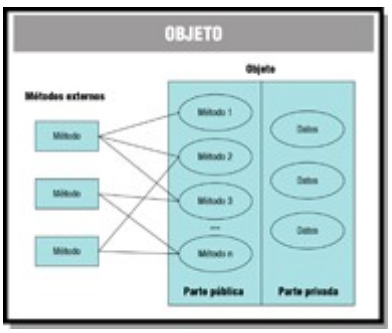


Por tanto, estudiando los objetos que están presentes en un problema podemos dar con la solución a dicho problema. Los objetos tienen unas características fundamentales que los distinguen:

- **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 el comportamiento serían acciones como: `arrancar(), parar(), acelerar(), frenar()` etc.

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 V**ariables** M**iembro**. 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.

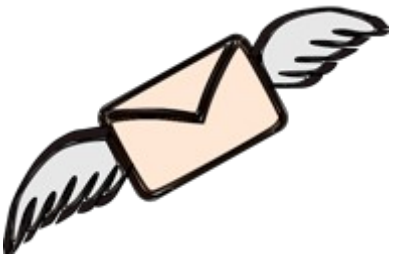
La única forma de manipular la información del objeto es a través de sus métodos. Es decir, si queremos saber el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo. De esta forma, evitamos que métodos externos puedan alterar los datos del objeto de manera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

> 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. ○

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 "copias" o "instancias" como necesitemos. Esas copias son los objetos de la clase.

> **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.

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.

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`**.

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. ○

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.

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. Esa 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.

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");
```

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:

4.5.- Destrucción de objetos y liberación de memoria.

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. ○

5.- Utilización de métodos.

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.

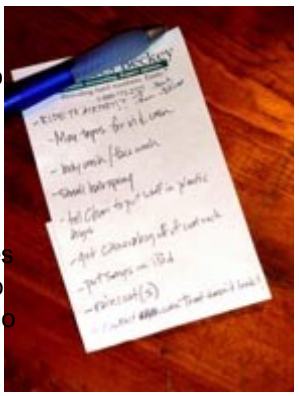
Para utilizar los métodos adecuadamente es conveniente conocer la estructura básica de que disponen.

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:

- **Inicializar** 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

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 utilizar el nombre del método, 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** del método, 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 definíamos 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.

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.

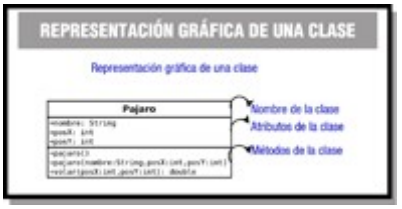
El método constructor tiene las siguientes particularidades:

- **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.

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.

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 estábamos 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:

- 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. ○

6.- Librerías de objetos (paquetes).

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.

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.

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**

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.

Si el proyecto lo hemos creado desde un entorno integrado como Netbeans, la compilación y ejecución se realizará al ejecutar la opción **RunFile (Ejecutar archivo)** o hacer clic sobre el botón **Ejecutar**.

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;
```

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
```

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**, así:

```
import java.util.Date;
```

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.

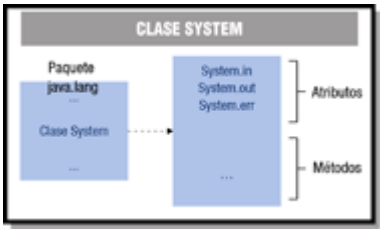
Los paquetes más importantes que ofrece el lenguaje Java son:

- `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.

7.- Programación de la consola: entrada y salida de la información.

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");
```

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**

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);
```

En el código anterior hemos creado un **InputStreamReader** a partir de **System.ini** 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.

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:

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 queramos escribir un mensaje y el valor de una variable debemos utilizar el operador de concatenación de cadenas (+), por ejemplo:

```
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.

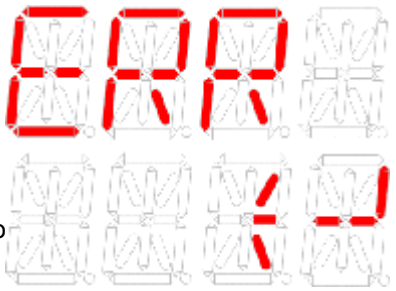
Por ejemplo, si queremos escribir el número **float** 12345.1684 con el punto de los miles y sólo dos cifras decimales la orden sería:

```
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.

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 estándar por pantalla");
System.err.println("Salida de error por pantalla");
```

La salida de este ejemplo en Netbeans es:

```
run:
Salida estándar por pantalla
Salida de error por pantalla
BUILD SUCCESSFUL (total time: 1 second)
```

Como vemos en un entorno como Netbeans, utilizar las dos salidas nos puede ayudar a una mejor depuración del código.

Autoevaluación

Relaciona cada clase con su función, escribiendo el número asociado a la función en el hueco correspondiente.

Clase.	Relación.	Función.
`Scanner`		1. Convierte los bytes leídos en caracteres.
`PrintStream`		2. Lee hasta un fin de línea.
`InputStreamReader`		3. Lee diferentes tipos de datos desde la consola de la aplicación.
`BufferedReader`		4. Contiene varios métodos para mostrar datos por pantalla.