

TEMA 3: UTILIZACIÓN DE OBJETOS

1.	INTRODUCCIÓN.....	1
2.	FUNDAMENTOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS.....	1
2.1	CONCEPTOS.....	2
2.2	BENEFICIOS.....	3
2.3	CARACTERÍSTICAS.....	4
2.4	LENGUAJES DE PROGRAMACIÓN ORIENTADOS A OBJETOS.....	5
3.	CLASES Y OBJETOS. CARACTERÍSTICAS DE LOS OBJETOS.....	6
3.1	PROPIEDADES Y MÉTODOS DE LOS OBJETOS.....	7
3.2	INTERACCIÓN ENTRE OBJETOS.....	7
3.3	CLASES.....	8
4.	UTILIZACIÓN DE OBJETOS.....	10
4.1	CICLO DE LA VIDA DE LOS OBJETOS.....	10
4.2	DECLARACIÓN.....	11
4.3	INSTANCIACIÓN.....	13
4.3.1.	CONCEPTO DE CONSTRUCTOR.....	14
4.4	REFERENCIAS A OBJETOS.....	16
4.5	MANIPULACIÓN.....	18
4.6	DESTRUCCIÓN DE OBJETOS Y LIBERACIÓN DE MEMORIA.....	23
4.7	OBJETOS STRING EN JAVA.....	23
5.	MÉTODOS.....	25
5.1	PARÁMETROS Y VALORES DEVUELTOS.....	26
5.2	USO DE MÉTODOS.....	28
5.3	DOCUMENTACIÓN DE UNA CLASE.....	31
5.4	OBJETOS INMUTABLES.....	34
5.5	COMPARACIÓN DE OBJETOS EN JAVA: MÉTODO EQUALS.....	37
5.6	MÉTODOS ESTÁTICOS.....	40
5.6.1.	EJEMPLOS: GENERAR NÚMEROS ALEATORIOS.....	43
5.6.2.	MÉTODOS “FÁBRICA” O PSEUDOCONSTRUCTORES.....	48
5.7	CLASES ENVOLTORIO EN JAVA.....	49
5.8	REPRESENTACIÓN TEXTUAL DE UN OBJETO EN JAVA: MÉTODO toString.....	50
6.	LIBRERÍA DE OBJETOS (PAQUETES).....	52
6.1	SENTENCIA IMPORT.....	53
6.2	COMPILAR Y EJECUTAR CLASES CON PAQUETES.....	55
6.3	JERARQUÍA DE PAQUETES.....	56

TEMA 3: UTILIZACIÓN DE OBJETOS

6.4	LIBRERIAS JAVA.....	57
7.	PROGRAMACIÓN DE LA CONSOLA: ENTRADA Y SALIDA DE LA INFORMACIÓN.....	58
7.1	CONCEPTOS SOBRE LA CLASE SYSTEM.....	58
7.2	ENTRADA POR TECLADO. CLASE SYSTEM.....	60
7.3	ENTRADA POR TECLADO. CLASE SCANNER.....	61
7.4	SALIDA POR PANTALLA.....	62
7.5	SALIDA DE ERROR.....	63
8.	MANIPULACIÓN DE LA FECHA Y LA HORA EN JAVA.....	64
8.1	LocalDate.....	64
8.2	LocalTime.....	66
8.3	LocalDateTime.....	67
8.4	Formateado de fechas.....	69
8.5	ChronoUnit.....	71
9.	EXCEPCIONES.....	73
9.1	CAPTURAR UNA EXCEPCIÓN CON TRY.....	75
9.2	EL MANEJO DE EXCEPCIONES CON CATCH.....	77

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 desarrollar, modificar y mantener, y por tanto, menos costosos.

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 o 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.

TEMA 3: UTILIZACIÓN DE OBJETOS

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 consustancial, 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, que es más cercana a nuestra forma de analizar, entender y resolver los problemas.

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 la 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, ¿qué quiere decir esto en la práctica?

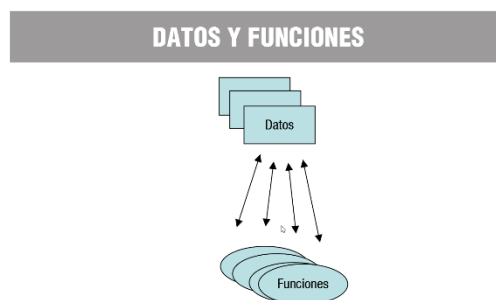
Pues que, a la hora de escribir un programa, nos fijaremos en los objetos involucrados, sus características comunes y las acciones que pueden realizar. Una vez localizados los objetos que intervienen en el problema real (espacio del problema), los tendremos que trasladar al programa informático (espacio de la solución). Con este planteamiento, la solución a un problema dado se convierte en una tarea sencilla y bien organizada.

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:

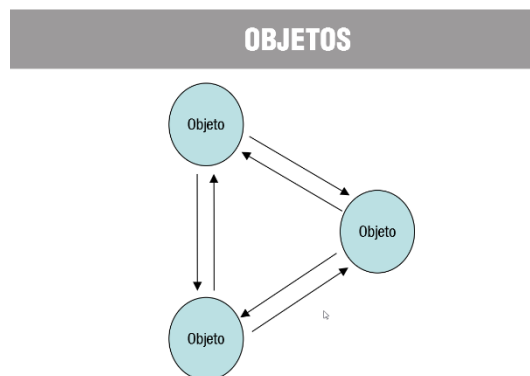
- Los programas se creaban y estructuraban de acuerdo con la arquitectura de la computadora donde se tenían que ejecutar.
- Al estar separados los datos de las funciones, éstos eran visibles en toda la aplicación. Ello ocasionaba que cualquier modificación en los datos podía requerir la modificación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

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**.

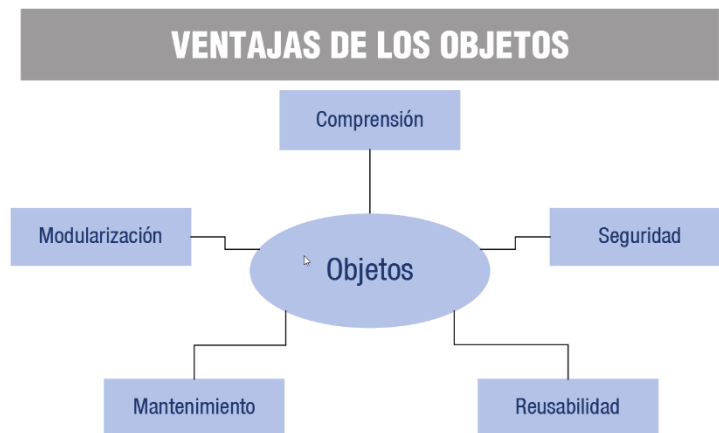
El concepto fundamental de la Programación Orientada a Objetos es, precisamente, los objetos. Pero ¿qué beneficios aporta la utilización de objetos?

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.** La definición 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.

TEMA 3: UTILIZACIÓN DE OBJETOS

- **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.

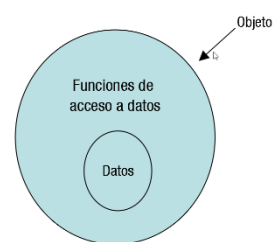
Cuando hablamos de **Programación Orientada a Objetos**, existen una serie de características que se deben cumplir. Cualquier lenguaje de programación orientado a objetos las debe contemplar. Las características más importantes del paradigma de la programación orientada a objetos son:

- **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 crea 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** maúllan, 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.

FUNCIONES Y DATOS



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 (MVC¹) 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.

¹ **MVC. Modelo-Vista-Controlador:** Modelo de programación que separa los datos de una aplicación, de la interfaz que los maneja y del controlador que define la forma en que actúa la interfaz.

TEMA 3: UTILIZACIÓN DE OBJETOS

- **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 James 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 C 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++.

3. CLASES Y OBJETOS. CARACTERÍSTICAS DE LOS OBJETOS.

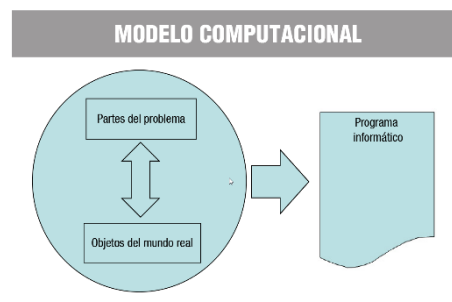
Al principio de la unidad veíamos que el mundo real está compuesto de objetos, y podemos considerar objetos casi cualquier cosa que podemos ver y sentir. Cuando escribimos un programa en un lenguaje orientado a objetos, debemos identificar cada una de las partes del problema con objetos presentes en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

En este contexto, un objeto 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 (número de bastidor).



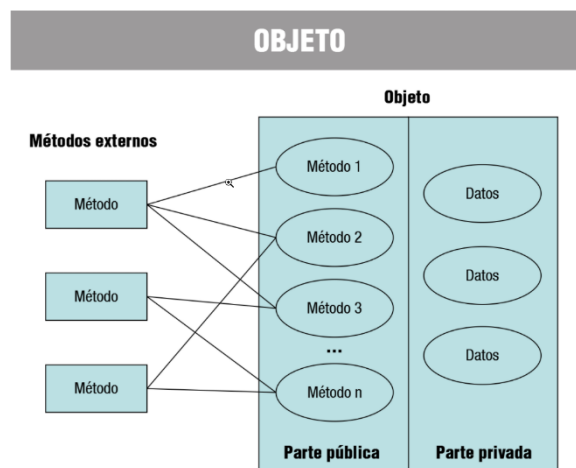
TEMA 3: UTILIZACIÓN DE OBJETOS

- **Estado.** El estado de un objeto viene determinado por una serie de propiedades o atributos que lo describen, y los valores de éstos. Por ejemplo, si tenemos un objeto **Vehículo**, 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 **Vehículo**, 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:

- **Atributos, campos o propiedades.** Son la 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 a su vez una **referencia a otro objeto**. Por ejemplo, un objeto de la clase **Vehículo** puede contener varios objetos de la clase **Rueda**.
- **Métodos o funciones miembro.** Son la 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. Esto significa que si queremos conocer el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo (si es que se dispone de un método para ello). De esta forma, evitamos que métodos externos al objeto puedan alterar sus datos de manera inadecuada. **Se dice que los atributos y los métodos están encapsulados dentro del objeto.**

3.2 INTERACCIÓN ENTRE OBJETOS.

¿Pero los distintos objetos de un programa pueden tener algún tipo de comunicación entre ellos?

Dentro de un programa los objetos se comunican llamando unos a los métodos de otros. 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 solicitud a un objeto para que realice una determinada acción.
Un método es la función o procedimiento al que se llama para actuar sobre un objeto (para enviarle un mensaje).

TEMA 3: UTILIZACIÓN DE OBJETOS

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 (de manera similar a como hacemos con variables de tipo entero, real, carácter, etc.).
- Comunicación entre los objetos mediante el envío de mensajes de 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 ordenador.

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

3.3 CLASES.

Hasta ahora hemos visto lo que son los objetos.

Piensa en el mundo real, en un tipo de objeto, una galleta y en el proceso industrial de producción de galletas. No necesitamos hacer y hornear una sola galleta, sino que lo normal será hacer y hornear un montón de ellas. Para realizar ese proceso, seguro que nos resulta útil disponer de un molde que establezca cómo va a ser cada objeto galleta. Pues bien, algo así ocurre en programación con los objetos, que usamos "molde" para construirlos.

Un programa informático se compone de muchos objetos, algunos de los cuales comparten la misma estructura y comportamiento. Si tuviéramos que definir la estructura y comportamiento del 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 en su interior una serie de métodos y variables.

TEMA 3: UTILIZACIÓN DE OBJETOS

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.

```
1  /*
2   * Estructura de una clase en Java
3   */
4
5  Cabecera de la clase
6  public class NombreClase { Cuerpo de la clase
7      // Declaración de los atributos
8
9      // Declaración de los métodos
10
11     public static void main (String[] args) {
12         // Declaración de variables y/o constantes
13
14         // Instrucciones del método
15     }
16
17
18 }
19
```

En las unidades anteriores ya hemos utilizado clases, aunque aún no sabíamos apenas nada sobre su significado exacto.

Por ejemplo, en la plantilla genérica que estamos usando para nuestros programas, en algunos ejemplos y en las tareas, hemos estado utilizando clases, todas ellas eran clases principales, sin ningún atributo y con un único método llamado **main**. Y así vamos a seguir por el momento, **en esta unidad no vamos a aprender a implementar clases sino a utilizarlas**. Usaremos las que nos proporciona la biblioteca de Java, también conocida como API (interfaz de programación de aplicaciones) de Java. También utilizaremos algún ejemplos de clase ya implementada que os proporcionemos como ejemplo.

Por ahora la única clase que vamos a implementar es la que contiene nuestro programa principal dentro de un **método main**. Ahí dentro estará todo nuestro programa.

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.

Por el momento será el único tipo de clase que vamos a implementar.

4. UTILIZACIÓN DE OBJETOS.

Una vez que un programador ha implementado una clase (por ejemplo todas las clases disponibles en la API de Java), 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 sinónimos. Es decir, nos referimos a objetos como instancias cuando queremos hacer hincapié que son casos concretos (instancias específicas creadas) 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 de nuevo en un **molde de galletas y en las propias 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, y todos los objetos de esa clase, aun siendo distintos, se parecerán bastante, ya que tendrán las mismas características generales.

Otro ejemplo, imagina que alguien ha implementado una clase **Persona** que reúne las características comunes de las personas (nombre, NIF, color de pelo, color de 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**.

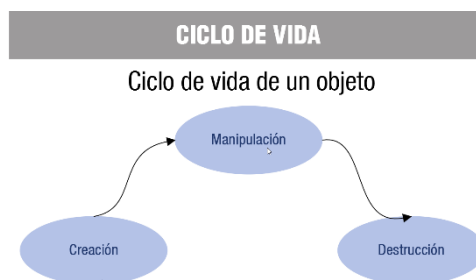
Cualquier objeto instanciado de una clase contiene una copia de todos los atributos definidos en la clase. En otras palabras, lo que está haciendo el operador **new** al instanciarlo es reservar el espacio necesario en la memoria del ordenador para guardar sus atributos y métodos, siguiendo para ello las indicaciones que le da la propia clase. 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**, o simplemente **instancias**. De igual forma, a los métodos que manipulan esas variables se les llama **métodos de instancia**.

En el ejemplo del objeto **trabajador**, las variables instancia serían **colorDePelo, peso, altura**, etc. Y los métodos de instancia serían **crecer, dormir, comer**, etc.

4.1 CICLO DE LA 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.



TEMA 3: UTILIZACIÓN DE 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**: indicar el tipo de objeto.
- **Instanciación**: creación del objeto utilizando el operador **new**.

Veamos primero cómo declarar un objeto. Para indicar el tipo de objeto debemos emplear la siguiente instrucción:

```
<Tipo> nombreObjeto;
```

donde:

- **Tipo** es la clase a partir de la cual se va a crear el objeto (nombre de la clase, que empezará por mayúscula), y
- **nombreObjeto** es el nombre de la variable referencia con la cual nos referiremos al objeto (empezará por minúscula).

Los tipos referenciados o referencias se utilizan para guardar una dirección de memoria en la que se encuentra un dato o un conjunto de datos.

Para entender mejor la declaración de objetos veamos un ejemplo.

La biblioteca de clases de la API Java ofrece un extenso conjunto de clases organizadas en paquetes que nos permiten programar con comodidad proporcionándonos muchas herramientas y utilidades que nosotros ya no vamos a tener que implementar. Un ejemplo podría ser la clase **Rectangle**, que representa a un rectángulo en el plano. Los objetos de esta clase representan rectángulos mediante una serie de datos (atributos). Entre esos atributos tenemos la **ubicación del rectángulo** (coordenadas (x,y) en el plano) y sus **dimensiones** (anchura y altura).

La clase **Rectangle** se encuentra dentro del paquete **java.awt**. Para poder disponer de esta clase en tu programa debes incluir la línea **import java.awt.Rectangle** en tu programa (después de la línea **package**, que probablemente sea tu primera línea de código):

```
import java.awt.Rectangle;
```

Si queremos declarar un objeto de tipo "rectángulo" a partir de la clase **Rectangle** lo haremos como hemos hecho hasta el momento con cualquier variable de tipo primitivo salvo que en este caso en lugar de usar un tipo primitivo (**int**, **double**, **char**, **boolean**, etc.) usaremos como tipo el nombre de la clase:

```
Rectangle rectangulo; // Declaración de una referencia
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Si queremos declarar varios, podemos hacerlo en líneas diferentes:

```
// Declaración de varias referencias (variables) cada una por separado
Rectangle r1;
Rectangle r2;
Rectangle r3;
```

O en la misma línea, como también hacemos con los tipos primitivos:

```
Rectangle r1, r2, r3; // Declaración de varias referencias (variables) en la misma línea
```

En Java, los nombres de las clases empiezan con mayúscula, como por ejemplo *Rectangle*, y los nombres de los objetos (variables) con minúscula, como por ejemplo *rectángulo*. De este modo sabemos rápidamente de qué tipo de elemento estamos hablando (una clase o un objeto).

Como puedes observar, poco se diferencia esta declaración de las declaraciones de variables que hacíamos para los tipos primitivos:

```
int numero;           // Declaración de una variable de tipo entero
Rectangle placa;      // Declaración de una variable de tipo referencia a un objeto de la clase Rectangle
```

Lo que sí es cierto es que **placa** aún no contiene una referencia a un objeto (una dirección a una zona de memoria donde se encuentren los atributos de una instancia de la clase **Rectangle**) de la misma manera que **numero** aún no contiene en su interior los bytes que representan a un número entero.

Al declarar una referencia, ésta se encuentra vacía hasta que se le asigne algún valor, como cualquier otra variable. Cuando una variable referencia a un objeto no contiene ninguna dirección a una instancia se dice que es una **referencia nula**, es decir, que contiene el valor **null** (también se dice que apunta a **null**). Esto significa que la variable referencia está creada, pero que aún no contiene una dirección de memoria a una instancia. Es decir, hemos preparado la referencia indicando a qué tipo de objetos va a apuntar, pero todavía no hemos reservado memoria para el contenido del objeto (para los atributos que alberga en su interior). Por tanto, aún no está apuntando a ninguna posición de memoria que contenga ese objeto (sus datos o atributos), y para que no haya confusión posible, la referencia apunta a un objeto inexistente llamado **null** (nulo).

Debes tener en cuenta que, **en Java, para una variable de tipo referencia recién creada, ni siquiera podríamos considerar que contiene inicialmente el valor null**. En este caso el compilador considera que aún no se le ha asignado ningún valor y si se intenta utilizar lo más probable es que se produzca un error de compilación. Se trata de algo similar a lo que sucedería por ejemplo con una variable de tipo **int**, recién declarada, que ni siquiera contiene el valor 0 sino que se considera "aún no inicializada" (se obtendría un error del tipo *variable might not have been initialized*).

En el ejemplo anterior lo más probable es que en **placa** haya un valor **null** (de la misma manera que en **número** haya un **0**), pero el compilador de Java, por seguridad, no nos dejará utilizar esas variables hasta que se les haya asignado explícitamente algún valor.

4.3 INSTANCIACIÓN.

Una vez creada una variable de tipo referencia a un objeto, debemos crear la instancia u objeto a la que va a apuntar esa referencia. Para ello utilizamos el operador **new** con la siguiente sintaxis:

```
nombreObjeto = new <ConstructorDeLaClase> ( [<parametro1>, <parametro2>, ..., <parametroN>] );
```

donde:

- **nombreObjeto** es el nombre de la variable referencia con la cual nos referiremos al objeto.
- **new** es el operador necesario para crear el objeto (invocar una llamada al constructor).
- **ConstructorDeLaClase** 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.
- **parametro1,...,parametroN**, 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 almacenar todos los valores de los atributos del objeto. De esta tarea se encarga automáticamente la máquina virtual de Java.

En el apartado anterior vimos cómo declarar un objeto de la clase **Rectangle** proporcionada por la biblioteca de clases de la API Java (paquete **java.awt**). Recuerda que poder disponer de esta clase en tu programa debes incluir la línea **import java.awt.Rectangle** al comienzo. Veamos ahora un ejemplo de creación de objetos instancias de la clase **Rectangle**.

Si queremos instanciar un objeto de tipo "rectángulo" a partir de la clase **Rectangle** tendremos que utilizar el constructor seguido de un paréntesis y unos valores para "alimentar" los parámetros con los que haya sido diseñado ese constructor. Por ejemplo:

```
Rectangle rectangulo; // Declaración
rectangulo = new Rectangle (0, 0, 10, 5); // Instanciación y asignación
```

Con la primera línea declaramos una variable de tipo "rectángulo" (en realidad una variable referencia a objetos de la clase **Rectangle**). Con la segunda estamos:

1. **ejecutando el constructor** de **Rectangle** para crear un objeto instancia de la clase **Rectangle**, cuyo vértice inferior izquierdo estará ubicado en las coordenadas (0,0), con una base de longitud 10 y una altura de longitud 5;
2. **asginando** a la variable **rectangulo** la referencia devuelta por el operador **new** tras la ejecución del constructor.

Como puedes observar, el operador **new** se encarga de devolver la referencia (algo así como una dirección de memoria) al objeto recién creado.

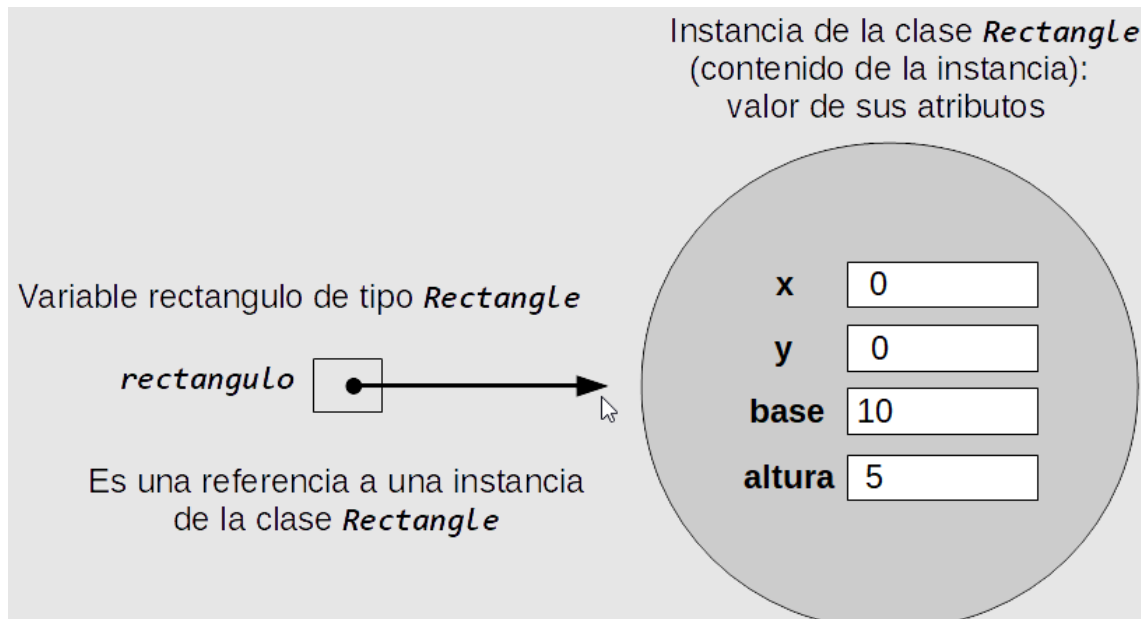
También podrías haberlo hecho todo en una única línea:

```
Rectangle rectangulo = new Rectangle (0, 0, 10, 5); // declaración + instaciación + asignación
```

TEMA 3: UTILIZACIÓN DE OBJETOS

En cualquier caso, tras la ejecución de estas instrucciones dispondrás de una variable llamada `rectangulo` que será una referencia que apuntará a un objeto instancia de la clase **Rectangle** cuyo contenido serán las coordenadas $x=0$, $y=0$, con una base de tamaño 10 y una altura de tamaño 5.

Gráficamente podríamos representarlo en abstracto con la siguiente figura:



4.3.1. CONCEPTO DE CONSTRUCTOR.

Acabamos de ver que para poder crear o "instanciar" un objeto de una clase determinada debemos utilizar el operador **new** seguido del nombre de la clase y un paréntesis con una opcional lista de parámetros. También hemos visto que el nombre del **constructor** de una clase con el nombre de la misma, de ahí la necesidad de poner paréntesis, y lo hemos descrito como un método especial que sirve para inicializar valores. En este apartado vamos a hablar 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 de forma explícita tras su ejecución, aunque implícitamente se devuelve el objeto que se está creando a través del operador **new**.

Para poder crear un objeto debemos instanciarlo utilizando el constructor de la clase. En el apartado anterior vimos un ejemplo de creación de un objeto de la clase **Rectangle** proporcionada por la biblioteca de clases de Java (paquete **java.awt**):

```
Rectangle rectangulo = new Rectangle (0, 0, 10, 5);
```

Con esa línea creábamos, a través de una invocación al constructor y mediante el uso del operador **new**, un **objeto instancia de la clase Rectangle** ubicado en la posición (0,0), de base 10 y de altura 5. Y a continuación se asignaba a la variable **rectangulo** la referencia devuelta por el operador **new** tras la ejecución del constructor.

La invocación a un constructor es lo primero que hay que hacer para poder disponer de un objeto y aplicar manipulaciones sobre él.

TEMA 3: UTILIZACIÓN DE OBJETOS

Un constructor tiene las siguientes características:

- **Es invocado automáticamente en la creación de un objeto**, y sólo esa vez;
- **Los nombres de los constructores no empiezan con minúscula**, como el resto de los métodos, ya que **se llaman igual que la clase** y los nombres de clase deben empezar siempre con letra mayúscula, según convenio;
- **Puede haber varios constructores** para una clase, que se llamarán todos igual, pero que se diferenciarán por su lista de argumentos (**sobrecarga**);
- El constructor puede tener **parámetros** para indicar con qué valores iniciar o configurar los atributos del objeto que se va a crear;
- En Java, **es necesario que toda clase tenga al menos un constructor**. Si no se implementa ningún constructor para una clase Java, **y solamente en ese caso**, el compilador crea un **constructor por omisión vacío**, que inicializa los atributos a sus valores por omisión, según del tipo que sean: **0** para los tipos numéricos, **false** para los **boolean** y **null** para las referencias.



Ejercicio Resuelto

Disponemos de las variables `r1`, `r2` y `r3`, de tipo `Rectangle`:

```
Rectangle r1, r2, r3;
```

Estas variables son referencias (direcciones de memoria) a objetos de la clase `Rectangle`. Inicialmente no contienen ningún valor. No lo harán hasta que no se les asigne el valor devuelto por una llamada al operador `new` al invocar a un **constructor** de la clase `Rectangle`.

Escribe el código necesario para crear **dos objetos de tipo rectángulo** (instancias de la clase `Rectangle`) y asignar su referencia (lo que devuelve el operador `new` tras invocarse al constructor) a las variables `r1` y `r2`:

- ✓ el **primer rectángulo** estará situado en la posición (1,1), tendrá una base de 5 y una altura de 3;
- ✓ el **segundo rectángulo** estará ubicado en la posición (2,0), tendrá una base de 2 y una altura de 2.

Recuerda que la ubicación del rectángulo se define a partir de las coordenadas (x,y) de su vértice inferior izquierdo.

Para crear dos rectángulos será necesario hacer **dos llamadas al constructor con el operador `new`**. El valor devuelto por `new` (la "referencia" o dirección de memoria) será lo que asignemos a las variables `r1` y `r2`:

```
Rectangle r1, r2, r3;  
r1= new Rectangle (1,1,5,3); // Ubicado en (1,1), base 5, altura 3  
r2= new Rectangle (2,0,2,2); // Ubicado en (2,0), base 2, altura 2
```

Sabemos que existen **varios constructores para la clase `Rectangle`**. Además del que acabamos de usar, también se dispone de uno con **sólo dos parámetros: base y altura**. En este caso la posición inicial del rectángulo se considera por omisión el origen de coordenadas (0,0). Teniendo esto en cuenta, instancia un tercer rectángulo de base 8 y altura 4 ubicado en (0,0) usando este segundo constructor y asigna su referencia a la variable `r3`, que aún no contiene ningún valor.

Este caso es más simple aún que los anteriores, pues el constructor que vamos a usar tiene menos parámetros:

```
r3= new Rectangle (8,4); // Ubicado en (0,0), base 8, altura 4
```

¿Podría haberse utilizado el **constructor anterior con cuatro parámetros**? Por supuesto que sí:

```
r3= new Rectangle (0,0,8,4); // Ubicado en (0,0), base 8, altura 4
```

Pero la idea era utilizar este otro constructor que nos resultaba más cómodo para este caso concreto (con sólo dos parámetros teníamos suficiente).

4.4 REFERENCIAS A OBJETOS.

Ya hemos visto que lo que contiene una variable de tipo referencia (variable que no es de tipo primitivo) es una referencia (algunos lo llaman puntero) a una zona de memoria donde se encuentran alojados los atributos de un objeto instancia de una clase.

Eso significa que en la variable de tipo "referencia" u "objeto" no está almacenado el objeto en sí (la sucesión de todos los bytes que representan la información que contienen los atributos), sino un número (que nosotros llamamos referencia, dirección de memoria o puntero) que indica donde están todos esos bytes con los valores de los atributos. Una analogía podría ser la de que la variable referencia es un casillero donde se guarda la llave de una habitación. La habitación con todo su contenido sería el objeto con sus atributos dentro y el casillero sería la variable de tipo referencia que contiene la dirección de memoria o referencia (llave de la habitación) a donde está el objeto (habitación).

Por tanto, cuando se realiza una asignación de un objeto de tipo referencia a otro objeto de tipo referencia, lo único que estamos haciendo es copiar y asignar referencias (direcciones de memoria o "llaves") pero no haciendo copias de los objetos ("habitaciones").

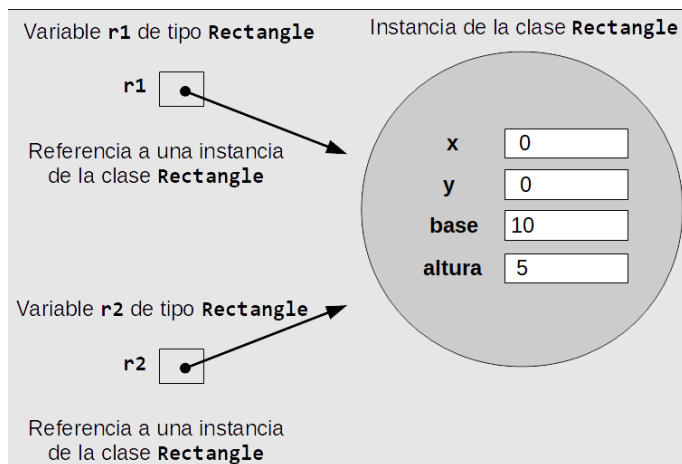
En este ejemplo que ya analizamos anteriormente:

```
Rectangle r1 = new Rectangle (0, 0, 10, 5); // declaración + instanciación + asignación
```

podíamos considerar que se llevaban a cabo tres pasos:

1. se declaraba la variable **r1** de tipo referencia a objetos instancias de la clase **Rectangle**,
2. se instanciaba un objeto de la clase **Rectangle** con las siguientes características: ubicación en las coordenadas (0,0), base 10, altura 5 (invocación al constructor a través del operador **new**);
3. se asignaba a la variable **r1** la referencia que devuelve el operador **new** tras ejecutar el código del constructor y reservarse una zona de memoria para almacenar los atributos de un nuevo objeto instancia de la clase **Rectangle**.

Es decir, que hemos hecho un **new** y por tanto tenemos un nuevo objeto **Rectangle**. Y como además hemos hecho una asignación a **r1**, podemos decir que **r1** "apunta" a ese objeto recién creado. Eso es más correcto que decir que **r1** "contiene" al objeto rectángulo. Por eso usábamos una representación gráfica de este tipo:

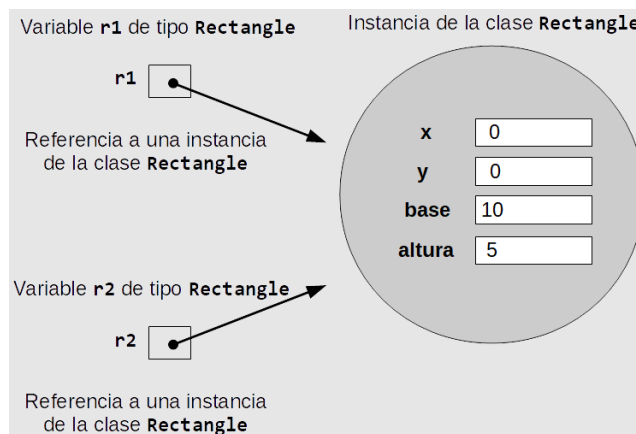


TEMA 3: UTILIZACIÓN DE OBJETOS

Si a continuación hacemos:

```
Rectangle r1= r1;
```

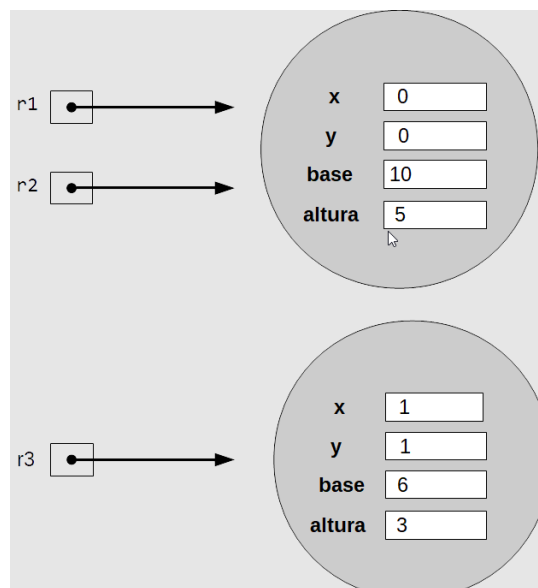
No estaremos creando un nuevo objeto Rectangle, sino que estaremos asignando a **r2** la misma referencia (posición de memoria o "llave") que contiene **r1**. De este modo, aunque **r1** y **r2** son dos variables distintas (podemos imaginarlas como dos casilleros distintos) están apuntando al mismo objeto (podríamos decir que ambos contienen copias de una misma llave para una misma habitación). Eso significa que cualquier acción que se lleve a cabo sobre el objeto apuntado por **r1** también se estará llevando a cabo sobre el objeto apuntado por **r2**, pues no son más que dos referencias al mismo objeto. No se trata de objetos diferentes sino de variables referencia diferentes que en este caso apuntan a la misma instancia (un objeto de la clase **Rectangle**).



Si queremos tener un segundo objeto de tipo **Rectangle**, tendríamos que hacer un segundo **new** (o bien, como veremos más adelante, una invocación a algún método que devuelva un nuevo objeto instancia de la clase **Rectangle**) y asignarlo a alguna variable de tipo referencia a la clase **Rectangle**. Por ejemplo:

```
Rectangle r3= new Rectangle (1, 1, 6, 3);
```

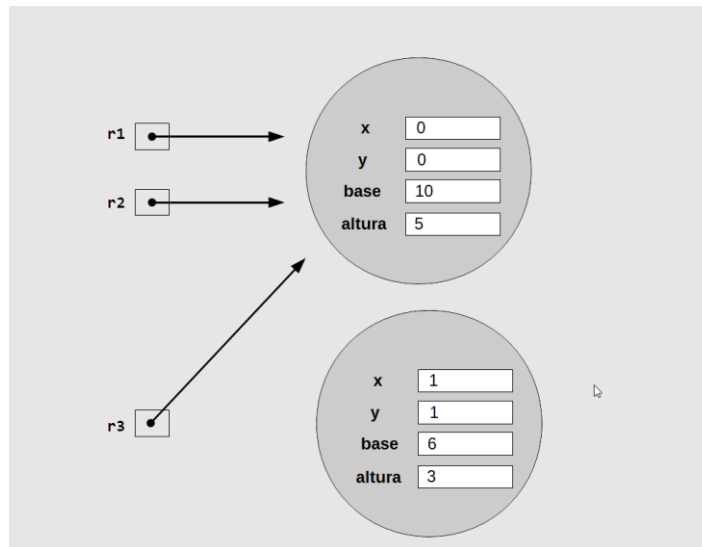
En este caso la variable **r3** apuntaría a un nuevo objeto de tipo **Rectangle** diferente al que apuntan **r1** y **r2**:



TEMA 3: UTILIZACIÓN DE OBJETOS

¿Qué sucede si en un momento dado **r3** deja de apuntar a su objeto original y se le asigna por ejemplo el valor de **r2**?

```
r3= r2; // r3 "apunta" a donde esté apuntando r2 (no hace una copia, "apuntan" o "referencian" al mismo objeto)
```



No habría problema, **r3** cambiará de valor y a partir de ese momento las variables **r1**, **r2** y **r3** apuntarían al mismo objeto. Por otro lado, el objeto al apuntaba originalmente **r3** se habrá quedado "perdido" o "huérfano" (no existe ninguna variable referencia que apunte a él). El recolector de basura de Java acabará borrándolo de la memoria, pues se trata de un elemento irrecuperable desde el programa y por tanto inútil.

PARA SABER MAS: En este otro enlace puedes ver un artículo también muy interesante acerca de cómo poder comparar dos objetos en Java: [Comparación](#)

4.5 MANIPULACIÓN.

Una vez creado e instanciado un objeto ¿cómo accedemos a su contenido o le enviamos un mensaje?

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:

```
nombreObjeto.mensaje
```

Por ejemplo, para acceder a los atributos o variables instancia del objeto se utiliza la siguiente sintaxis:

```
nombreObjeto.nombreAtributo
```

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis:

```
nombreObjeto.nombreMetodo ( lista de parámetros )
```

TEMA 3: UTILIZACIÓN DE OBJETOS

La lista de parámetros consiste en una lista de valores (puede ser una lista vacía) que cada método espera que se le pasen.

Ahora bien, para poder acceder a cualquier miembro de un objeto (ya sea un atributo o un método) debe de tratarse de un **miembro accesible**, es decir, que tengamos permiso para acceder a él. Si no tenemos acceso a ese miembro desde fuera del objeto es como si no existiera. De hecho, ni siquiera sabremos que existe, pues será un elemento estructural de la clase que como usuarios de ella desconoceremos. En ese sentido podemos imaginar la clase como una "caja negra" que define con qué miembros del objeto (atributos y/o métodos) nos podemos comunicar. Cuando escribamos programas que utilicen objetos de una determinada clase, no todo el contenido de esos objetos será accesible directamente desde nuestro código mediante el operador punto. Sólo tendremos acceso a aquellos elementos a los que se nos permita acceder. Son los elementos que definen la **interfaz de la clase**. A esos elementos también se les suele llamar **miembros públicos** del objeto.

Para entender mejor cómo se pueden manipular objetos vamos a continuar con nuestro ejemplo de los objetos de la clase **Rectangle**. En primer lugar instanciamos el objeto invocando al **método constructor, que se llama igual que la clase**, a través del operador **new** e indicando los parámetros correspondientes a la posición y a las dimensiones del rectángulo como ya hemos hecho en anteriores ocasiones:

```
Rectangle rectangulo = new Rectangle(50, 50, 150, 150);
```

La sentencia anterior declara la variable **rectangulo** como una referencia de tipo **Rectangle**, y hace que apunte a un nuevo objeto que se crea al invocar al constructor con el operador **new**. El constructor recibe como parámetros las coordenadas de la esquina superior izquierda, que sería el punto **(50, 50)** en la pantalla, y el ancho y alto del mismo (longitud de la base y la altura), que sería 150 en ambos casos.

Una vez que hemos instanciado un objeto de tipo **Rectangle** y que disponemos de una variable referencia a él llamada **rectangulo**, si queremos cambiar el valor de los atributos utilizamos el operador punto. Por ejemplo, para cambiar la dimensión del rectángulo, para que tanto su altura como su base sean 100, en lugar de los 150 que le habíamos asignado, podría hacerse modificando directamente el valor de sus atributos así:

```
rectangulo.height = 100 ;
```

```
rectangulo.width = 100 ;
```

Esto hemos podido hacerlo porque los atributos **height** (altura) y **width** (anchura o base) de la clase **Rectangle** son **públicos** (accesibles desde fuera del objeto) y por tanto manipulables desde nuestro programa. Si no, no habríamos podido acceder a ellos.

Otra posibilidad habría sido utilizar un método para hacer algo similar a lo anterior. Por ejemplo, la siguiente línea de código fija el tamaño del rectángulo a 200 por 200, tanto de base como de altura. El método utilizado se llama **setSize**:

```
rectangulo.setSize(200,200) ;
```

Lo más habitual será proceder de esta segunda manera, pues normalmente los atributos suelen ser privados dentro del objeto y no se pueden manipular desde fuera. Para ello se nos proporcionarán métodos (como por ejemplo **setSize**) que nos permitirán llevar a cabo esas manipulaciones pero de forma "controlada".

TEMA 3: UTILIZACIÓN DE OBJETOS

Aquí tienes un ejemplo completo donde se prueba todo lo anterior:

```
// Instanciamos un nuevo objeto de la clase Rectangle
// y apuntamos a él mediante la variable r1
// Ubicación en (0,0) y dimensiones 10 de base, 5 de altura
System.out.print  ("Creando objeto instancia de la clase Rectangle ");
System.out.println ("y referenciado desde la variable r1.");
Rectangle r1= new Rectangle (0,0, 10, 5);
System.out.println ();

// Mostramos los atributos o propiedades del objeto al que apunta r1
System.out.println ("Rectángulo r1:");
System.out.println ("Ubicación: x=" + r1.x + " y=" + r1.y);
System.out.println ("Dimensiones: base= " + r1.width+ " altura= " + r1.height);

// Modificamos los atributos o propiedades del objeto al que apunta r1

// 1. Accediendo directamente a sus atributos y modificándolos (pues son públicos)
r1.height= 20;
r1.width= 100;
System.out.println ();
System.out.println ("Rectángulo r1 modificado mediante manipulación de atributos:");
System.out.println ("Ubicación: x=" + r1.x + " y=" + r1.y);
System.out.println ("Dimensiones: base= " + r1.width+ " altura= " + r1.height);

// 2. Utilizando un método para modificar algunos de sus atributos: setSize
r1.setSize(200, 50);
System.out.println ();
System.out.println ("Rectángulo r1 modificado mediante llamada a un método:");
System.out.println ("Ubicación: x=" + r1.x + " y=" + r1.y);
System.out.println ("Dimensiones: base= " + r1.width+ " altura= " + r1.height);
```

La salida debería ser algo similar a lo siguiente:

Creando objeto instancia de la clase Rectangle y referenciado desde la variable r1.

Rectángulo r1:
Ubicación: x=0 y=0
Dimensiones: base= 10 altura= 5

Rectángulo r1 modificado mediante manipulación de atributos:
Ubicación: x=0 y=0
Dimensiones: base= 100 altura= 20

Rectángulo r1 modificado mediante llamada a un método:
Ubicación: x=0 y=0
Dimensiones: base= 200 altura= 50



Ejercicio Resuelto

Disponemos de tres variables de tipo referencia a objetos de la clase `Rectangle`: `r1`, `r2`, `r3`.

```
Rectangle r1, r2, r3;
```

Instancia un objeto `Rectangle` con ubicación en las coordenadas (10,10), de base 10 y altura 5. Asigna a la variable `r1` la referencia a ese objeto recién creado.

Mostrar retroalimentación

Bastaría con llamar al **constructor** de la clase utilizando el operador `new`:

```
r1= new Rectangle (10,10, 10, 5);
```

Asigna a la variable `r2` la misma referencia a la que apunta `r1`. Es decir, que `r1` y `r2` hagan referencia al mismo objeto. Muestra por pantalla el contenido de los atributos de los objetos a los que apuntan `r1` y `r2` (deberían ser el mismo objeto y por tanto tener los mismos valores).

TEMA 3: UTILIZACIÓN DE OBJETOS

Bastaría con asignar a `r2` el valor actual de `r1`:

```
r2= r1;
```

Ahora podríamos decir que `r1` y `r2` apuntan al mismo objeto. Por tanto, cualquier modificación que se realice en `r2` a través de los miembros del objeto en realidad se está haciendo también en `r1`. En realidad la modificación no se realiza ni en `r1` ni en `r2`, sino en el objeto al que apuntan ambos, que es el mismo. **Recuerda que en tanto en `r1` como en `r2` no se guarda realmente el objeto sino una referencia (un número, también conocido como dirección de memoria o puntero) al objeto.**

Para mostrar el contenido de sus atributos podríamos hacer algo así como:

```
// Mostramos los atributos o propiedades del objeto al que apunta r1
System.out.println ("Rectángulo r1:");
System.out.println ("Ubicación: x=" + r1.x + " y=" + r1.y);
System.out.println ("Dimensiones: base= " + r1.width+ " altura= " + r1.height);
System.out.println ();

// Mostramos los atributos o propiedades del objeto al que apunta r2
System.out.println ("Rectángulo r2:");
System.out.println ("Ubicación: x=" + r2.x + " y=" + r2.y);
System.out.println ("Dimensiones: base= " + r2.width+ " altura= " + r2.height);
System.out.println ();
```

Y la salida debería ser algo así como:

```
Rectángulo r1:
Ubicación: x=10 y=10
Dimensiones: base= 10 altura= 5

Rectángulo r2:
Ubicación: x=10 y=10
Dimensiones: base= 10 altura= 5
```

Modifica la altura del rectángulo al que apunta `r2` poniéndola al valor 6. Recuerda que el atributo público para consultar y modificar la altura de un objeto de la clase `Rectangle` es `height`.

A continuación muestra por pantalla de nuevo el valor de los atributos de los objetos a los que apuntan tanto `r1` como `r2`. ¿Qué modificaciones observas?

Para modificar la altura del rectángulo al que apunta `r2` basta con modificar el valor el atributo `height`:

```
r2.height= 6;
```

Si mostramos en pantalla de nuevo el valor de los atributos de `r1` y `r2` deberíamos observar algo como:

```
Rectángulo r1:
Ubicación: x=10 y=10
Dimensiones: base= 10 altura= 6

Rectángulo r2:
Ubicación: x=10 y=10
Dimensiones: base= 10 altura= 6
```

Es decir, que las modificaciones realizadas sobre el objeto al que hace referencia `r2` también han tenido efecto sobre el objeto al que apunta `r1` pues ambas variables "apuntan" (o "hacen referencia") a la misma instancia (objeto) en memoria.

Instancia ahora un segundo objeto `Rectangle` con ubicación en las coordenadas (0,0), de base 100 y altura 20 asignando a la variable `r3` la referencia a ese objeto recién creado. Utiliza la versión del constructor que sólo requiere dos parámetros (base y altura).

A continuación, muestra por pantalla el valor de los atributos el objeto al que apunta `r3`.

TEMA 3: UTILIZACIÓN DE OBJETOS

Para crear un nuevo objeto `Rectangle` con menos datos iniciales basta con utilizar el constructor apropiado. En esta ocasión utilizamos uno que sólo recibe dos parámetros (`width` y `height`), sin coordenadas. Este constructor establecerá las coordenadas por omisión (0,0).

```
r3= new Rectangle (100, 20);
```

La información que tendría aparecer por pantalla debería ser similar a la siguiente:

```
Rectángulo r3:  
Ubicación: x=0 y=0  
Dimensiones: base= 100 altura= 20
```

Modifica ahora el valor de la variable `r3` asignándole el valor de la variable `r2`. A continuación modifica la base y la altura de `r3` (en realidad del objeto rectángulo al que apunta `r3`) al valor 50 en ambos casos. Por último muestra por pantalla el contenido de los atributos de los objetos a los que apuntan `r1`, `r2` y `r3`. ¿Qué conclusiones puedes obtener de lo que observas por la pantalla?

Para dar el nuevo valor a `r3` bastaría con asignar a `r3` el valor actual de `r2`:

```
r3= r2;
```

Para modificar la base y altura del rectángulo al que apunta `r3` podemos el método `setSize`:

```
r3.setSize(50,50);
```

Aunque también podríamos haberlo hecho mediante la asignación directa a los atributos de base y altura.

Por último, si mostramos por pantalla el valor de los atributos de los objetos a los que apuntan cada una de las variables referencia `r1`, `r2` y `r3`, deberíamos obtener algo similar a lo siguiente:

```
Rectángulo r1:  
Ubicación: x=10 y=10  
Dimensiones: base= 50 altura= 50  
  
Rectángulo r2:  
Ubicación: x=10 y=10  
Dimensiones: base= 50 altura= 50  
  
Rectángulo r3:  
Ubicación: x=10 y=10  
Dimensiones: base= 50 altura= 50
```

¿Qué conclusiones podemos extraer de aquí? Está claro que ahora mismo **las tres variables apuntan al mismo objeto en memoria** (el primero que se instanció y al que apuntaba inicialmente sólo `r1`) y por tanto **cualquier manipulación que hagamos desde cualquier de las tres va tener efecto sobre ese mismo objeto** y va a ser reflejado de igual manera desde esas tres variables mientras sigan conteniendo la misma referencia (apuntando al mismo objeto). En el momento en que alguna de ellas apunte a otro objeto, cualquier modificación que se realice ya no afectará a las otras.

¿Qué crees que acabará sucediendo a medio/largo plazo con el segundo objeto que se ha creado y al que apuntaba inicialmente `r3`?

Mostrar retroalimentación

Dado que ya no hay ninguna variable que apunte a ese objeto (no está "referenciado" desde ninguna variable de tipo referencia), el recolector de basura acabará descubriendo que se trata de un objeto inaccesible y por tanto considerándolo como "basura" en el sistema, de manera que lo borrará liberando el espacio ocupado para que pueda ser reutilizado. ¿Cuándo lo borrará? Cuando el recolector de basura lo considere oportuno teniendo en cuenta la carga del sistema en cada momento.

4.6 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 que puedan ser reutilizados por el sistema. A esta acción se le denomina **destrucción del objeto**.

En Java la destrucción de objetos corre a cargo del **recolector de basura (garbage collector)**. ¡Un gran invento!, ya que nos permite como programadores prácticamente desentendernos de ese asunto, sabiendo además que nunca van a producirse problemas por olvidarse de recoger la basura adecuadamente, como ocurría en otros lenguajes.

El recolector de basura es un sistema de destrucción automática de objetos que ya no son utilizados (no existe ninguna variable de tipo referencia que enlace a ellos). Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador **new**, pero que ya ha dejado de estar referenciada, es decir, ya no hay forma de llegar a ella, y por tanto ya no es posible volver a usar ese objeto. Esto evita que al programar tengamos que preocuparnos de realizar la liberación de memoria.

El recolector de basura se ejecuta en 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, cuando el procesador está menos ocupado.

4.7 OBJETOS STRING EN JAVA.

Cuando veíamos los tipos de datos primitivos, indicábamos que Java proporcionaba un tipo de dato especial para los textos o cadenas de caracteres que era el tipo de dato **String**. Realmente este tipo de dato es un **tipo referenciado**, no un tipo primitivo. Para declarar una variable de este tipo de dato llamada por ejemplo **mensaje**, debemos escribir lo siguiente:

```
String mensaje;
```

Con lo que ya sabemos ahora podemos observar que **String** es realmente una clase que nos proporciona Java para facilitar el trabajo con cadenas de texto a partir de la cual creamos nuestro objeto llamado **mensaje**.

En Java, los nombres de las clases empiezan con mayúscula, como por ejemplo *String*, y los nombres de los objetos con minúscula, como por ejemplo *mensaje*. De este modo sabemos rápidamente de qué tipo de elemento estamos hablando (una clase o un objeto).

Como puedes observar, 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 una referencia a un objeto de la clase **String**, pero en la práctica la declaración no es diferente a como podría ser por ejemplo la declaración de una variable de tipo **int**.

```
int numero;      // Declaración de una variable de tipo entero
String mensaje;  // Declaración de una variable de tipo referencia a un objeto de la clase String
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Lo que sí es cierto es que **mensaje** aún no contiene una referencia a un objeto (una dirección a una zona de memoria donde se encuentren los atributos de una instancia de la clase **String**) de la misma manera que **numero** aún no contiene en su interior los bytes que representan a un número entero. Lo más probable es que en **mensaje** haya un valor **null** (de la misma manera que en **numero** haya un **0**), pero el compilador de Java, por seguridad, no nos dejará usar esas variables hasta que se les haya asignado explícitamente algún valor.

Para instanciar un objeto **String** debemos invocar a su constructor, como ya hemos visto con ejemplos de otros objetos:

```
String texto = new String();
```

Así estaríamos instanciando el objeto **texto**. Para ello utilizaríamos el operador **new** y el constructor de la clase **String**, 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 una versión del constructor que contenga parámetros. Por ejemplo:

```
texto = new String ("El primer programa");
```

Cadena vacía

Valor para una cadena que consiste en una cadena con 0 caracteres. Sería algo así como el "cero" de las cadenas. Es representada habitualmente por dos comillas dobles "" sin nada en medio.

Aunque String es una clase, se trata de una clase muy especial.

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, aunque también puede hacerse. Por tanto las sentencias `texto=new String("El primer programa");` y `texto="El primer programa";` son totalmente equivalentes. Lo que hace Java cuando se encuentra una del segundo (sin invocación al constructor con new) tipo es "traducirla" al primer tipo (con new y el constructor).

Queremos llamar también tu atención sobre el hecho de que no es lo mismo decir que una referencia de tipo String apunte a la cadena vacía (objeto String de longitud cero, que no tiene ningún carácter) que decir que apunte a null (que no es un String, sino una forma de indicar que no apunta a ningún objeto). La cadena vacía sí es un objeto. El valor null no representa a ningún objeto sino a la referencia nula (es decir, que no se está apuntando a nada).

Como ya hemos visto con otras variables de tipo referencia (y con los tipos primitivos), la **declaración y asignación de un valor inicial** a una variable de tipo **String** puede hacerse también en una misma línea o instrucción. Es decir, que podemos hacer:

```
String texto = new String ("El primer programa"); // Declaración + instanciación + asignación de valor inicial
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Donde se llevan a cabo las siguientes operaciones y en este orden:

1. declaración de la variable **texto**, de tipo **String** (o de la clase **String**);
2. instanciación de un objeto de la clase **String** mediante la llamada a su constructor con el parámetro "el primer programa" a través del operador **new**;
3. asignación de la referencia al objeto recién creado (lo que devuelve **new**) a la variable **texto** para que a partir de ese momento la variable **texto** "referencie" o "apunte" a un objeto de la clase **String** con el contenido "el primer programa".

Ahora bien, dado que Java nos ofrece la comodidad de poder instanciar objetos **String** sin necesidad de usar el operador **new** para invocar al constructor, lo normal es que la inicialización de un objeto de tipo **String** normalmente tenga el siguiente aspecto:

```
String texto = "El primer programa"; // Declaración + instanciación + asignación de valor inicial
```

Donde se llevan a cabo las mismas acciones que en el ejemplo anterior.

5. MÉTODOS.

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Contienen las operaciones que se pueden sobre el estado objeto (sobre el valor de sus atributos), y que son ejecutadas cuando el método es invocado.

Para utilizar los métodos adecuadamente es conveniente conocer la estructura de su cabecera pues será la interfaz de comunicación con el objeto al cual pertenecen los métodos.

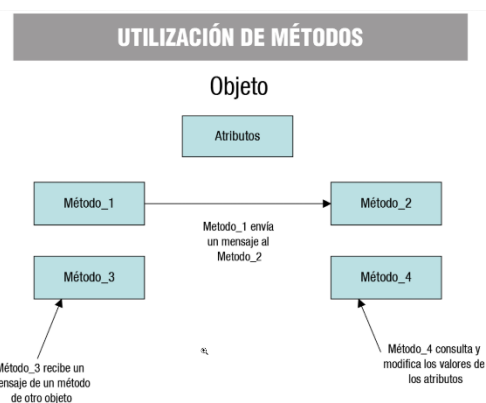
Los métodos están compuestos por una **cabecera** y un **cuerpo**. La cabecera puede tener modificadores, como por

ejemplo **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 fragmento de código externo al objeto. Veremos más adelante los modificadores que se pueden usar y lo que significa cada uno.

Respecto al cuerpo de un método, contendrá el código de la acción a realizar pudiéndose encontrar en su interior acciones del tipo:

- **inicialización** de atributos del objeto,
- **consulta** de valores de atributos,
- **modificación** y manipulación de los valores de algunos atributos,
- **llamada** a otros métodos, del mismo objeto o de otros objetos externos,
- **ejecución** de sentencias que contengan estructuras de control como condiciones o bucles para realizar cálculos,

Si te fijas, el cuerpo del método podría considerarse como un "miniprograma" que realiza una serie de acciones cuando es invocado. Por ahora nosotros no vamos a escribir el código Java que hay en el interior de un método. Nos vamos a limitar a utilizar los métodos de los objetos que vamos a instanciar dentro de nuestros programas sin preocuparnos sobre cómo han sido implementados por dentro. Eso ha sido la tarea de otro programador y a nosotros nos llegan las clases listas para que podamos utilizarlas. Es decir que podamos instanciar objetos a partir de esas clases usando sus constructores y realizar operaciones sobre esos objetos a través de las llamadas a sus métodos.



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, es decir, de sus atributos, se lleva a cabo mediante la **lista de parámetros**.

En general, los parámetros de un método pueden pasarse de dos formas diferentes:

- **por valor.** El valor de los parámetros no cambia al finalizar la ejecución 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 tiene 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, todos parámetros se pasan por valor.

Ahora bien, en el caso de parámetros que son referencias a objetos debemos tener en cuenta que:

1. al pasarse por valor la referencia al objeto (zona de memoria a la que se apunta), ésta no podrá ser modificada desde el interior del método (siempre se "apuntará" al mismo objeto o zona de memoria);
2. el estado del objeto al que se apunta (sus atributos) sí podría ser modificado si se le aplica algún método que altera alguno de sus atributos;
3. sólo si el objeto al que se apunta es inmutable podemos garantizar que ese objeto no puede ser modificado dentro de un método.

Respecto a la interfaz de un método en Java, debemos tener en cuenta que:

- **indica qué tipo de valor devuelve** (o bien **void** o tipo "vacío" en el caso de no devolver nada). 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 la ausencia de valor de retorno;
- **dispone de una lista de argumentos o parámetros.** 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. Esos argumentos serían como las "variables de entrada" del método pues lo habitual es que un método no haga peticiones de datos por teclado sino que reciba ya sus datos de entrada a través de sus parámetros.

El *valor de retorno* es la información que devuelve un método tras su ejecución.

TEMA 3: UTILIZACIÓN DE OBJETOS

La cabecera de un método en Java se declara como sigue:

```
public tipoDeDatoDevuelto nombreMetodo (listaDeParametros) // Cabecera del método {  
    // Cuerpo del método (algo a lo que no tiene acceso quien use el método)  
}
```

Como puedes observar, 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:

```
(tipoParametro1 nombreParametro1, ..., tipoParametroN nombreParametroN)
```

Aquí tienes un ejemplo real de la cabecera de un método que pertenece a la clase **Rectangle**:

```
public boolean contains(int x, int y)
```

En este caso el valor de retorno es de tipo **boolean** y la lista de parámetros contiene dos parámetros ambos de tipo **int**.

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 (lista de parámetros actuales) debe coincidir en número, tipo y orden con los **parámetros** incluidos en la declaración del método (lista de parámetros formales), ya que de lo contrario se produciría un error de sintaxis.

Un ejemplo de llamada al método anterior podría ser el siguiente:

```
boolean test= r1.contains(3,4);
```

donde:

1. **r1** debería apuntar a un objeto instancia de la clase **Rectangle**;
2. la invocación al método devolvería un valor **true** o **false** dependiendo de que el punto (3,4) se encuentre dentro del área abarcada por el rectángulo representado por ese objeto.

En este caso los **parámetros formales** serían **x** e **y**, de tipo **int**, y los **parámetros actuales** serían **3** y **4**. Podríamos decir que durante la ejecución del método para esa llamada en concreto, en el interior del código del método la variable local o parámetro **x** pasará a valer 3 y la variable **y** pasará a valer 4.

5.2 USO DE MÉTODOS.

Una vez que hemos visto el aspecto de la interfaz de un método, vamos a ver algunos ejemplos de cómo utilizar algunos de ellos. Para ello vamos a volver a la clase **Rectangle** que ya hemos usado en varias ocasiones. Podemos observar en la siguiente tabla algunos ejemplos de los métodos que incorpora:

Algunos métodos de la clase `Rectangle`

Método	Descripción
<code>public boolean contains(int x, int y)</code>	Indica si el rectángulo contiene al punto (x,y). Devuelve un <code>boolean</code> .
<code>public Rectangle intersection(Rectangle r)</code>	Calcula el rectángulo resultante de realizar la intersección con el rectángulo <code>r</code> que se pasa como parámetro. Devuelve una referencia a un nuevo objeto <code>Rectangle</code> .
<code>public boolean intersects(Rectangle r)</code>	Indica si el objeto rectángulo y el rectángulo <code>r</code> que se pasa como parámetro tienen algún punto en común (hacen intersección). Devuelve un <code>boolean</code> .
<code>public void setLocation(int x, int y)</code>	Se desplaza la ubicación del rectángulo a las coordenadas (x,y) pasadas como parámetros. No devuelve ningún valor.
<code>public void setSize(int width, int height)</code>	Se establece un nuevo tamaño para el rectángulo a partir de la nueva anchura (base) y altura pasadas como parámetros. No devuelve nada.
<code>public Rectangle union(Rectangle)</code>	Calcula un nuevo rectángulo resultado de realizar la unión con el rectángulo <code>r</code> que se pasa como parámetro. Devuelve una referencia a un nuevo objeto <code>Rectangle</code> .
<code>public double getHeight()</code>	Obtiene la altura actual del objeto rectángulo. Devuelve un <code>double</code> .
<code>public double getWidth()</code>	Obtiene la anchura (longitud de la base) actual del objeto rectángulo. Devuelve un <code>double</code> .
<code>public double getX()</code>	Obtiene la posición x del rectángulo (esquina inferior izquierda). Devuelve un <code>double</code> .
<code>public double getY()</code>	Obtiene la posición y del rectángulo (esquina inferior izquierda). Devuelve un <code>double</code> .

La forma en la que estos métodos han sido implementados "por dentro" (el cuerpo o "interior" de los métodos) es algo que por el momento no nos interesa. Los objetos que instanciamos de la clase **Rectangle** son algo así como "cajas cerradas" con las cuales nos podemos comunicar a través de sus métodos sin conocer su mecanismo interior. Su implementación interna ha sido la labor de otro programador o programadora que se encargó de codificar y probar apropiadamente cada uno esos métodos y que nosotros vamos a utilizar sin preocuparnos acerca de cómo están desarrollados internamente. A ese efecto nos podemos considerar como "usuarios" de los objetos de esta clase, pues no tenemos acceso a su código fuente ni es nuestra misión conocer su interior. Nos vamos a limitar a usarlos.

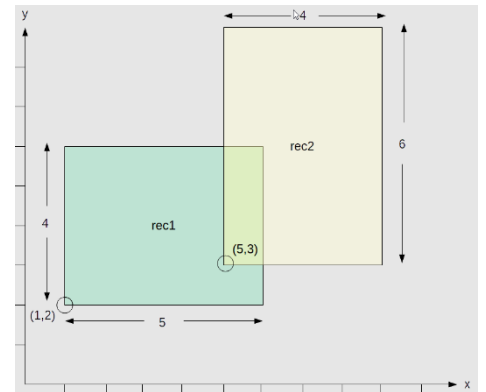
Cuando se nos proporcione una clase que no hemos desarrollado nosotros (por ejemplo las de la biblioteca de la API de Java), lo único que debemos conocer de ella es su "interfaz", es decir, aquellos miembros de la clase que sean públicos.

Esta interfaz consistirá en una lista atributos públicos (a veces ninguno) y una lista de métodos y constructores donde se indicará para cada uno con qué parámetros debe ser "alimentado" y qué tipo devuelve (su interfaz).

TEMA 3: UTILIZACIÓN DE OBJETOS

Vamos ahora a ver algunos ejemplos de uso los métodos anteriores dados un par de objetos de tipo **Rectangle** recién creados:

```
Rectangle rec1= new Rectangle (1,2, 5, 4);  
Rectangle rec2= new Rectangle (5,3, 4, 6);
```



1.- Comprobamos si los rectángulos contienen el punto (3,3):

Para ello podemos utilizar el método **contains**:

```
// Comprobamos si los rectángulos contienen al punto (3,3)  
System.out.println ("rec1 contiene el punto (3,3): " + (rec1.contains(3,3) ? "sí":"no") );  
System.out.println ("rec2 contiene el punto (3,3): " + (rec2.contains(3,3) ? "sí":"no") );
```

```
rec1 contiene el punto (3,3): sí  
rec2 contiene el punto (3,3): no
```

2.- Comprobamos si ambos rectángulos tienen alguna zona en común (su intersección es no nula):

Para ello podemos utilizar el método **intersects**:

```
// Comprobamos si ambos rectángulos tienen alguna zona en común (su intersección es no nula)  
System.out.println ("rec1 tiene alguna zona en común con rec2: " +  
    (rec1.intersects(rec2) ? "sí":"no"));
```

```
rec1 tiene alguna zona en común con rec2: sí
```

Si hubiéramos hecho la operación a la inversa: **rec2.intersects(rec1)**, ¿habríamos obtenido el mismo resultado?

3. Calculamos el área en común de ambos rectángulos (un nuevo rectángulo):

```
// Calculamos el área en común de ambos rectángulos (un nuevo rectángulo "intersección")  
Rectangle rec3= rec1.intersection(rec2);
```

Observa que en este caso estamos obteniendo una tercera instancia de la clase **Rectangle** (un nuevo objeto) sin utilizar el constructor.

Pueden obtenerse objetos instancia de una clase a través de métodos que no sean el constructor.

Si quieres observar los atributos de este nuevo rectángulo, basta con mostrarlos en pantalla:

```
// Mostramos los atributos de ese nuevo rectángulo  
System.out.println ("Rectángulo rec3 intersección de rec1 y rec2:");  
System.out.println ("Ubicación: x=" + rec3.x + " y=" + rec3.y);  
System.out.println ("Dimensiones: base= " + rec3.width+ " altura= " + rec3.height);
```

```
Rectángulo rec3 intersección de rec1 y rec2:  
Ubicación: x=5 y=3  
Dimensiones: base= 1 altura= 3
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Y en este caso, si hubiéramos realizado la operación a la inversa: **rec1.intersects(rec2)**, ¿se habría obtenido también el mismo resultado?

4.- Reubicamos el rectángulo rec1 en la posición (2,2):

Para ello podemos utilizar el método **setLocation**:

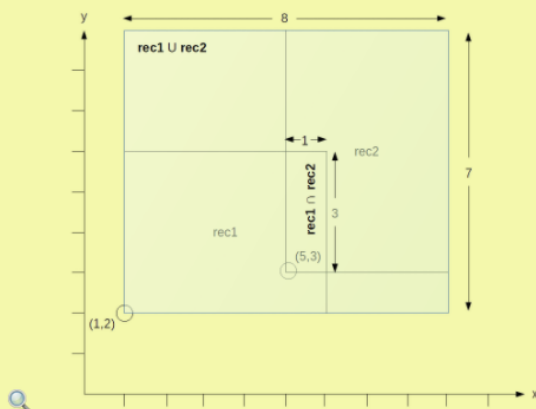
```
// Reubicamos el rectángulo rec1 en la posición (2,2)
rec2.setLocation(2, 2);
```



Ejercicio Resuelto

Dados los dos rectángulos anteriores **rec1** y **rec2**, objetos de la clase **Rectangle**, vamos a seguir probando algunos de métodos.

Calcula el rectángulo que se obtendría como **unión** de ambos rectángulos **rec1** y **rec2**:



El rectángulo resultante debería tener las siguientes características:

```
Rectángulo unión de rec1 y rec2:
Ubicación: x=1 y=2
Dimensiones: base= 8 altura= 7
```

Mostrar retroalimentación

Para ello podemos usar el operador **union**, obteniendo una referencia a un nuevo objeto instancia de la clase **Rectangle** que habrá que almacenar en alguna variable de tipo referencia a **Rectangle**:

```
// Calculamos la unión área ocupada por ambos rectángulos (un nuevo rectángulo "unión")
Rectangle recUnion= rec1.union (rec2);
```

Y mostramos por pantalla los atributos de ese nuevo objeto:

```
// Mostramos los atributos de ese nuevo rectángulo
System.out.println ("Rectángulo rec3 unión de rec1 y rec2:");
System.out.println ("Ubicación: x=" + recUnion.x + " y=" + recUnion.y);
System.out.println ("Dimensiones: base= " + recUnion.width+ " altura= " + recUnion.height);
```

Fíjate que éste es un nuevo caso en el que la aplicación de un método a un objeto tiene como resultado la obtención de un nuevo objeto que estará ubicado en otra zona de memoria y que por tanto necesitamos una nueva variable de tipo referencia para poder "apuntar" a él. Con esto podemos observar que **la invocación al constructor con el operador new no tiene por qué ser la única forma de obtener nuevos objetos sino que pueden existir métodos que también devuelven referencias a nuevas instancias.**

5.3 DOCUMENTACIÓN DE UNA CLASE.

Una vez que ya hemos estado manipulando objetos a través del uso de sus métodos, es posible que te preguntes:

- ¿cómo puedo conocer todas las operaciones (métodos) que puedo realizar sobre un objeto que sea instancia de una determinada clase? ¿qué valores devuelven estos métodos? ¿qué parámetros hay que pasarles para que funcionen correctamente?
- ¿tienen los objetos de esa clase algunos atributos públicos a los que pueda acceder?
- ¿cuántos constructores tiene esa clase? ¿qué parámetros tienen?

Normalmente cuando se distribuye un conjunto de clases para que pueda ser utilizada por otros programadores, además de los archivos binarios que contienen esas clases también se proporciona una documentación sobre ese paquete o conjunto de clases donde puedes encontrar las respuestas a todas esas preguntas.

Dependiendo del lenguaje (y a veces incluso del entorno), dispondremos de un sistema de documentación u otro. Pero en general se tratará de un conjunto de documentos donde para cada clase se ofrecerá una descripción de todos y cada uno de sus miembros públicos para que esa clase pueda ser utilizada correcta y apropiadamente por cualquier programador sin que tenga por qué conocer ningún detalle de su implementación interna.

En el caso de Java, se utiliza un sistema de documentación conocido como **javadoc** que proporciona toda la información disponible sobre cada clase en formato HTML facilitando la navegación entre clases, paquetes, métodos, etc. Esto quiere decir que si vamos a utilizar una biblioteca de clases, además de las propias clases (archivos binarios) deberían también proporcionarnos el javadoc de esa biblioteca para que sepamos de qué clases disponemos, qué métodos tienen, qué hacen, qué parámetros necesitan, que valores se devuelven y en qué circunstancias, etc.

Puedes echar un primer vistazo a toda esta documentación consultando el siguiente enlace: [Documentación javadoc de la API de Java](#)

Aquí podrás observar una lista de los paquetes básicos de la API de Java. Dado que ahora mismo todo eso te puede parecer una "sobredosis" de información, límitate a pulsar sobre el enlace al módulo **java.desktop** y a continuación sobre el enlace al paquete **java.awt**. Una vez ahí te aparecerá una tabla resumen con una lista de interfaces ("*Interface Summary*") y a continuación otra tabla con una lista de clases ("*Class Summary*"). Ahí podrás observar todas las clases del paquete **java.awt**, entre ellas tendrás la clase **Rectangle**. Pulsa sobre ella para navegar por la documentación de esa clase, que al menos ya has utilizado.

Si te has perdido entre tanta información, también puedes pulsar sobre el siguiente enlace: [Documentación javadoc de la API de Java](#)

que te llevará directamente a la documentación sobre la clase **Rectangle**.

Una vez allí, te encontrarás con una página con la siguiente estructura:

- **cabecera**, sección "**Class Rectangle**", donde te aparecerá el título "**Class Rectangle**" (nombre de la clase). Aquí encontrarás abundante información general sobre la clase, la mayoría de la cual aún no entenderás. Pero sí observarás que, entre otras cosas, dispones de una **descripción de la clase** explicando para qué puede servir y qué características tendrán los objetos que se instancien a partir de ella;

TEMA 3: UTILIZACIÓN DE OBJETOS

Module `java.desktop`

Package `java.awt`

Class `Rectangle`

`java.lang.Object`
`java.awt.geom.RectangularShape`
`java.awt.geom.Rectangle2D`
`java.awt.Rectangle`

All Implemented Interfaces:

`Shape`, `Serializable`, `Cloneable`

Direct Known Subclasses:

`DefaultCaret`

```
public class Rectangle
extends Rectangle2D
implements Shape, Serializable
```

A `Rectangle` specifies an area in a coordinate space that is enclosed by the `Rectangle` object's upper-left point (`x`, `y`) in the coordinate space, its width, and its height.

A `Rectangle` object's width and height are public fields. The constructors that create a `Rectangle`, and the methods that can modify one, do not prevent setting a negative value for width or height.

A `Rectangle` whose width or height is exactly zero has location along those axes with zero dimension, but is otherwise considered empty. The `isEmpty()` method will return true for such a `Rectangle`. Methods which test if an empty `Rectangle` contains or intersects a point or rectangle will always return false if either dimension is zero. Methods which combine such a `Rectangle` with a point or rectangle will include the location of the `Rectangle` on that axis in the result as if the `add(Point)` method were being called.

- sección "***Nested Class Summary***", que por ahora no entendemos ni necesitamos consultar;
- sección "***Field Summary***", donde podrás observar los **atributos, propiedades** o "**campos**" públicos (*fields* en inglés) de la clase;

Field Summary

Fields		
Modifier and Type	Field	Description
int	<code>height</code>	The height of the <code>Rectangle</code> .
int	<code>width</code>	The width of the <code>Rectangle</code> .
int	<code>x</code>	The X coordinate of the upper-left corner of the <code>Rectangle</code> .
int	<code>y</code>	The Y coordinate of the upper-left corner of the <code>Rectangle</code> .

- sección "***Constructor Summary***", con una tabla resumen de los **constructores** disponibles para esta clase;

Constructor Summary

Constructors	
Constructor	Description
<code>Rectangle()</code>	Constructs a new <code>Rectangle</code> whose upper-left corner is at (0, 0) in the coordinate space, and whose width and height are both zero.
<code>Rectangle(int width, int height)</code>	Constructs a new <code>Rectangle</code> whose upper-left corner is at (0, 0) in the coordinate space, and whose width and height are specified by the arguments of the same name.
<code>Rectangle(int x, int y, int width, int height)</code>	Constructs a new <code>Rectangle</code> whose upper-left corner is specified as (x,y) and whose width and height are specified by the arguments of the same name.
<code>Rectangle(Dimension d)</code>	Constructs a new <code>Rectangle</code> whose top left corner is (0, 0) and whose width and height are specified by the <code>Dimension</code> argument.
<code>Rectangle(Point p)</code>	Constructs a new <code>Rectangle</code> whose upper-left corner is the specified <code>Point</code> , and whose width and height are both zero.
<code>Rectangle(Point p, Dimension d)</code>	Constructs a new <code>Rectangle</code> whose upper-left corner is specified by the <code>Point</code> argument, and whose width and height are specified by the <code>Dimension</code> argument.
<code>Rectangle(Rectangle r)</code>	Constructs a new <code>Rectangle</code> , initialized to match the values of the specified <code>Rectangle</code> .

TEMA 3: UTILIZACIÓN DE OBJETOS

- sección "**Method Summary**", con otra table resumen de los **métodos públicos** de esta clase;

Method Summary		
All Methods	Instance Methods	Concrete Methods
Deprecated Methods		
Modifier and Type	Method	Description
void	<code>add(int newX, int newY)</code>	Adds a point, specified by the integer arguments newX, newY to the bounds of this Rectangle.
void	<code>add(Point pt)</code>	Adds the specified Point to the bounds of this Rectangle.
void	<code>add(Rectangle r)</code>	Adds a Rectangle to this Rectangle.
boolean	<code>contains(int x, int y)</code>	Checks whether or not this Rectangle contains the point at the specified location (x,y).
boolean	<code>contains(int X, int Y, int W, int H)</code>	Checks whether this Rectangle entirely contains the Rectangle at the specified location (X,Y) with the specified dimensions (W,H).
boolean	<code>contains(Point p)</code>	Checks whether or not this Rectangle contains the specified Point.
boolean	<code>contains(Rectangle r)</code>	Checks whether or not this Rectangle entirely contains the specified Rectangle.

- sección "**Field Detail**", donde se explica con más detalle cada uno de los atributos de la clase;

Field Detail

x

```
public int x
```

The X coordinate of the upper-left corner of the Rectangle.

Since:

1.0

See Also:

`setLocation(int, int), getLocation()`

- sección "**Constructor Detail**", donde se explica detalladamente el **funcionamiento de cada uno de los constructores** de la clase;

Rectangle

```
public Rectangle(int x,
                 int y,
                 int width,
                 int height)
```

Constructs a new Rectangle whose upper-left corner is specified as (x,y) and whose width and height are specified by the arguments of the same name.

Parameters:

x - the specified X coordinate

y - the specified Y coordinate

width - the width of the Rectangle

height - the height of the Rectangle

Since:

1.0

TEMA 3: UTILIZACIÓN DE OBJETOS

- sección "**Method Detail**", donde se explica detalladamente el **funcionamiento de cada uno de los métodos** de la clase.

Method Detail

getX

```
public double getX()
```

Returns the X coordinate of the bounding Rectangle in double precision.

Specified by:

[getX](#) in class [RectangularShape](#)

Returns:

the X coordinate of the bounding Rectangle.

Aquí tendrás toda la información que pudieras necesitar para usar esa clase. Como puedes observar, la tabla con la descripción de sus métodos y constructores puede ser como diez veces más grande que la pequeña tabla resumen con algunos ejemplos de métodos que nosotros hemos usado en la sección anterior para resolver algunos ejercicios. Ahora se trata de una documentación "real" y es exhaustiva, pues contiene toda la información que pudiera requerir un programador que quisiera hacer uso de esa clase en sus programas.

5.4 OBJETOS INMUTABLES.

Se dice que una es **clase immutable** cuando sus instancias una vez creadas e inicializadas no pueden modificar su estado, es decir, que son **objetos inmutables**. Esto significa que todos los métodos que llevan a cabo operaciones que en principio alterarían el estado interno de estos objetos, en realidad lo que hacen es generar una nueva instancia con ese nuevo estado.

Algunos ejemplos de clases en Java cuyos objetos son inmutables son:

- la clase **String**;
- las clases envoltorio o "*wrappers*" (**Byte, Integer, Integer, Long, Float, Double, Character, Boolean**);
- las clases para manipulación de fecha y hora: **LocalDate, LocalTime, LocalDateTime**.

Esto significa que cada vez que se llame a un método de un objeto instancia estas clases que implique algún tipo de modificación en su estado, no se va a llevar a cabo ese cambio en el propio objeto sino que se devolverá un nuevo objeto con esos cambios.

Un caso típico de clase inmutable en Java es la clase **String**. Para objetos instancias esta clase, métodos como **concat**, **replace** o **trim** no modifican el estado del objeto original sino que generan un nuevo objeto con la modificación que resulta de la aplicación del método. Si probamos el siguiente ejemplo:

TEMA 3: UTILIZACIÓN DE OBJETOS

```
// Declaramos y creamos una nueva cadena (declaración + instanciación + asignación)
String cadenaOriginal = new String ("Hola");

// Mostramos su contenido en pantalla
System.out.println("Cadena original: " + cadenaOriginal);

// Le concatenamos la cadena " caracola" (aplicación del método concat)
cadenaOriginal.concat(" caracola");

// Volvemos a mostrar su contenido en pantalla
System.out.println(cadenaOriginal);
```

```
Cadena original: Hola
Cadena tras la aplicación del método concat: Hola
```

Es decir, que la concatenación no ha afectado al objeto **cadenaOriginal**, sino que se ha generado (instanciado) un nuevo objeto de la clase **String** con el resultado de esa modificación (una nueva cadena cuyo valor será "Hola caracola"). Si quisiéramos haber visto el resultado de la modificación en pantalla deberíamos haber mostrado directamente el resultado de la operación:

```
// Declaramos y creamos una nueva cadena
String cadenaOriginal = new String ("Hola");

// Mostramos su contenido en pantalla
System.out.println("Cadena original: " + cadenaOriginal);

// Mostrar el resultado de aplicar el método concat
System.out.println("Cadena tras la aplicación del método concat: " + cadenaOriginal.concat(" caracola"));
```

O bien asignar a una segunda variable de tipo referencia a **String** ese resultado y a continuación mostrar su contenido por pantalla:

```
// Declaramos y creamos una nueva cadena
String cadenaOriginal = new String ("Hola");

// Mostramos su contenido en pantalla
System.out.println("Cadena original: " + cadenaOriginal);

// Declaramos una segunda variable para referenciar objetos de la clase String
String cadenaTransformada;

// Y le asignamos el resultado de aplicar el método concat sobre el cadenaOriginal
cadenaTransformada= cadenaOriginal.concat(" caracola");

// Mostramos el contenido de ese segundo objeto
System.out.println("Cadena tras la aplicación del método concat: " + cadenaTransformada);
```

En ambos casos deberíamos obtener una salida del tipo:

```
Cadena original: Hola
Cadena tras la aplicación del método concat: Hola caracola
```

TEMA 3: UTILIZACIÓN DE OBJETOS

La conclusión que obtenemos de este experimento es que métodos de manipulación de este tipo de objetos no afectan al objeto sobre el que se aplica la operación sino que generan un nuevo objeto como resultado de la operación aplicada. En este caso se dice que se trata de **objetos inmutables** y, por extensión, se habla de **clases inmutables**.

En estos casos necesitamos una segunda referencia (una variable) para poder almacenar la referencia al nuevo objeto que se ha creado como resultado de la operación. Si no almacenamos en una variable la referencia al objeto devuelta por el método que se ha invocado para llevar a cabo la operación, esa instancia quedaría sin referenciar desde nuestro programa y el recolector de basura acabará eliminándola.

Si queremos utilizar solamente una variable, podríamos hacerlo volviendo a asignar a la misma variable referencia a **String** en la que se almacenaba el contenido original, la referencia al nuevo objeto creado tras la operación de concatenación:

```
// Declaramos, instanciamos y asignamos una nueva cadena
String miVar = new String("Hola");

// Mostramos su contenido en pantalla
System.out.println("Cadena inicial: " + miVar);

// Y le ahora asignamos a la misma variable el resultado de aplicar el método concat sobre la cadena inicial
miVar = miVar.concat(" Antonio");

// Mostramos el nuevo contenido de la referencia, que ahora apunta a un nuevo objeto String
System.out.println("Cadena tras la aplicación del método concat: " + miVar);
```

En este caso la salida por pantalla sería idéntica, pero habríamos perdido la referencia a objeto **String** original, pues en esa variable hemos almacenado ahora la referencia al nuevo objeto **String** resultado del a concatenación.

Ahora bien, dadas las particularidades de la clase **String** en Java, lo usual será que:

1. no encuentres el uso de **new** con una llamada al constructor para crear nuevas cadenas (objetos instancia de la clase **String**);
2. no veas el uso del método **concat** para concatenarlas, sino que te encontrarás directamente con el operador **=** (asignación) para la instanciación (llamada al constructor) y asignación, así como el operador **+** para concatenar un objeto **String** con otro objeto **String**.

Teniendo en cuenta eso, el anterior fragmento de código quedaría de la siguiente manera:

```
// Declaramos, instanciamos y asignamos una nueva cadena
// (llamada implícita al constructor sin necesidad de new ni de escribir el constructor)
String miVar = "Hola";

// Mostramos su contenido en pantalla
System.out.println("Cadena inicial: " + miVar);

// Y le asignamos el resultado de aplicar el operador de concatenación + sobre la cadena inicial
miVar = miVar + " Antonio";

// Mostramos el contenido de ese segundo objeto
System.out.println("Cadena tras la aplicación del método concat: " + miVar);
```

El resultado en este caso también debería ser el mismo.

Con las **clases inmutables**, el estado de un objeto no se modifica (objeto inmutable), sino que cada intento de modificación del estado de un objeto (llamada a un método que implique cambios) genera como resultado una nueva instancia.



Debes conocer

En el siguiente vídeo se explica gráficamente lo que se acaba de exponer más arriba.

Clases inmutables



Ver en la plataforma

[Resumen del vídeo](#)

Resumen del vídeo: Se inicia con el título: Clases inmutables.

Una clase inmutable es aquella cuyas instancias, una vez inicializadas, no pueden modificarse. Por ejemplo: las clases envoltorio (como `Character` o `Integer`) o la clase `String` son inmutables. Se muestra un ejemplo de qué sucede en la memoria cuando hacemos asignaciones en instancias de estas clases.

Supongamos que en un programa declaramos una variable `miVar` de tipo `String`. Se ubicará en algún sitio de la memoria, supongamos por ejemplo `0x001`, podría ser cualquier otra.

Ahora, al ejecutarse la instrucción: `miVar = "Hola";`, `miVar` hará referencia (apuntará) a la dirección donde está el `String` "Hola". La instrucción: `miVar = miVar + "Antonio";` provocará que `miVar` haga referencia a la dirección donde está el `String` "Hola Antonio". Y ahora, el espacio de memoria `0x00F` queda marcado para que el recolector de basura lo elimine cuando se estime oportuno.

5.5 COMPARACIÓN DE OBJETOS EN JAVA: MÉTODO `EQUALS`.

En algunas ocasiones necesitarás comparar dos objetos para saber si son iguales. Cuando comparábamos dos variables de tipo primitivo (reales, enteros, caracteres, booleanos, etc.) teníamos muy claro qué significaba que dos elementos sean iguales. Sin embargo, cuando hablamos de objetos, quizá la cosa no esté tan clara.

Dado que un objeto suele estar caracterizado por un conjunto de valores que conforman su estado (atributos o propiedades), ¿cuál es el criterio para decidir si dos objetos son iguales o no? Por ejemplo podríamos decidir que dos objetos instancias de la misma clase son iguales si tienen el mismo estado (mismos valores en cada uno de sus atributos comparados uno a uno). Eso parece bastante razonable y es lo que se suele hacer.

Por ejemplo, para el caso de objetos de la clase **`Rectangle`** que hemos estado utilizando, podríamos decir que dos rectángulos son iguales si coinciden los valores de sus atributos de **ubicación (x, y)**, **base (width)** y **altura (height)**. ¿Y cómo llevamos a cabo esa comparación? ¿debemos nosotros escribir el código que vaya comparando uno a uno cada uno de sus atributos? Obviamente no. La idea es disponer de algún mecanismo de comparación que nos facilite esa labor de la misma manera que hacíamos con los tipos primitivos cuando utilizábamos el operador relacional `==` para obtener **true** si dos valores del mismo tipo eran iguales o **false** si no lo eran.

En Java ese mecanismo de comparación se realiza mediante el método **`equals`**. Todo objeto en Java dispone de un método **`equals`** que permite compararlo con otro objeto del mismo tipo. Veamos un ejemplo de comparación de varios objetos instancia de la clase **`Rectangle`**:

TEMA 3: UTILIZACIÓN DE OBJETOS

```
// Declaración de tres referencias a objetos Rectangle
// Instanciación de tres objetos Rectangle mediante el constructor
// Y asignación de las referencias devueltas por el operador new a cada una de
// las tres variables declaradas
Rectangle r1= new Rectangle (1,2, 5, 4);
Rectangle r2= new Rectangle (5,3, 4, 6);
Rectangle r3= new Rectangle (5,3, 4, 6);

// Comparación de los tres objetos mediante el uso del método equals
// sobre las variable referencia que apuntan a los objetos
System.out.println ("Ejemplos de comparación de objetos rectángulo:");
System.out.println ("Comparación r1 con r2 -> r1.equals(r2): " + r1.equals(r2));
System.out.println ("Comparación r1 con r3 -> r1.equals(r3): " + r1.equals(r3));
System.out.println ("Comparación r2 con r3 -> r2.equals(r3): " + r2.equals(r3));
```

```
Ejemplos de comparación de objetos rectángulo:
Comparación r1 con r2 -> r1.equals(r2): false
Comparación r1 con r3 -> r1.equals(r3): false
Comparación r2 con r3 -> r2.equals(r3): true
```

Lo cual es razonable dado que **r2** y **r3** tienen exactamente los mismos valores en sus atributos (son rectángulos idénticos), mientras que **r1** es diferente.

Del mismo modo que el operador **==** para tipos primitivos cumplía la **propiedad conmutativa** (daba igual qué valor estuviera a la izquierda o a la derecha), en el caso del método **equals** también sucede lo mismo: vamos a obtener el mismo resultado tanto si hacemos **r1.equals(r2)** como si hacemos **r2.equals(r1)**.

Es posible que ahora te preguntes, ¿y no podrá haber utilizado directamente el operador **==** para llevar a cabo esas comparaciones en lugar de tener que usar el método **equals**? Te retamos a que lo pruebes tú mismo:

```
System.out.println ("Comparación r1 con r2 -> r1 == r2: " + (r1 == r2) );
System.out.println ("Comparación r1 con r3 -> r1 == r3: " + (r1 == r3) );
System.out.println ("Comparación r2 con r3 -> r2 == r3: " + (r2 == r3) );
```

¿Qué has obtenido?

Lo más probable es que todas las comparaciones te hayan salido como **false**, ¿verdad? ¿Por qué? Porque lo que estás comparando con el operador **==** es el contenido de las variables referencia **r1**, **r2** y **r3** y no el contenido de los valores de los atributos de cada uno de los objetos a los que apuntan esas referencias. Cada una de esas variables apunta a una zona de memoria diferente (un objeto diferente) y por tanto contendrán un valor (dirección de memoria) diferente haciendo que su comparación de igualdad resulte como **false**.

El método **equals** sirve para evitar precisamente eso: para comprobar si dos objetos son iguales (tienen los mismos valores en sus atributos), independientemente de que sean objetos diferentes. Sin embargo el operador **==** está comprobando si las referencias son las mismas (mismo valor para la referencia o dirección de memoria), es decir si se apunta desde dos variables referencia al mismo objeto.

Si en el ejemplo anterior, realizamos la siguiente asignación:

```
r1= r3;
```

y a continuación volvemos a repetir las comparaciones anteriores:

```
System.out.println ("Comparación de objetos rectángulo:");
System.out.println ("Comparación r1 con r2 -> r1.equals(r2): " + r1.equals(r2));
System.out.println ("Comparación r1 con r3 -> r1.equals(r3): " + r1.equals(r3));
System.out.println ("Comparación r2 con r3 -> r2.equals(r3): " + r2.equals(r3));
System.out.println ();
System.out.println ("Comparación de referencias a objetos rectángulo:");
System.out.println ("Comparación r1 con r2 -> r1 == r2: " + (r1 == r2) );
System.out.println ("Comparación r1 con r3 -> r1 == r3: " + (r1 == r3) );
System.out.println ("Comparación r2 con r3 -> r2 == r3: " + (r2 == r3) );
```


TEMA 3: UTILIZACIÓN DE OBJETOS

deberíamos obtener los siguientes resultados:

```
Comparación de objetos rectángulo:  
Comparación r1 con r2 -> r1.equals(r2): true  
Comparación r1 con r3 -> r1.equals(r3): true  
Comparación r2 con r3 -> r2.equals(r3): true  
  
Comparación de referencias a objetos rectángulo:  
Comparación r1 con r2 -> r1 == r2: false  
Comparación r1 con r3 -> r1 == r3: true  
Comparación r2 con r3 -> r2 == r3: false
```

En este caso las comparaciones entre objetos (uso del método **equals**) devuelven todas **true**, pues **r1** y **r3** en realidad apuntan al mismo rectángulo y el rectángulo al que apunta **r2** tiene los mismos valores en sus atributos que el rectángulo al que apuntan **r1** y **r3**. Por tanto todos los rectángulos son iguales. Podríamos tener la "ilusión" de que existen tres objetos rectángulo aunque en realidad existen sólo dos para los cuales hay dos variables referencia que apuntan a uno de ellos (**r1** y **r3**) y una variable referencia que apunta al otro (**r2**).

Sin embargo las comparaciones entre las propias referencias (uso del operador **==**) algunas devuelven **true** y otras devuelven **false**:

- si comparamos **r1** y **r3** obtenemos **true** pues ambas variables almacenan la misma dirección de memoria o referencia (una referencia al mismo objeto rectángulo);
- si comparamos **r2** con alguna de las otras dos obtenemos **false**, pues **r2** contiene una referencia (dirección de memoria) diferente ya que apunta a otro objeto rectángulo.

En conclusión:

- el método **equals** nos indica si dos referencias apuntan a objetos que son iguales (tienen los mismos valores, aunque sean objetos distintos);
- el operador **==** nos indica si dos referencias apuntan al mismo objeto.

Lo habitual será por tanto que utilicemos el método **equals**.

En Java, siempre que queramos comparar dos objetos debemos utilizar el método `equals`.

5.6 MÉTODOS ESTÁTICOS.

Un **método estático** puede ser usado directamente desde la clase sin necesidad de hacer referencia a ninguna instancia.

Estos métodos también son conocidos como **métodos de clase**, frente a los **métodos de objeto** o instancia (es necesario un objeto para poder invocarlos).

Los métodos estáticos no pueden manipular instancias (objetos) y suelen ser utilizados para realizar operaciones genéricas independientes de las posibles instancias que puedan existir de esa clase en un momento concreto. De hecho estos métodos pueden ser invocados sin necesidad de que existan objetos de la clase.

La manera correcta de invocar a un método estático es escribiendo el nombre de la clase, seguido por el operador punto (.) más el nombre del método estático:

```
NombreClase.nombreMetodoEstatico();
```

En alguna ocasión, es posible que te encuentres con programas donde se haya realizado la llamada a un método estático escribiendo el nombre del objeto, seguido por el operador punto (.) más el nombre del método estático. Esta forma de invocar a un método estático no es apropiada. ¿Por qué? Porque aunque en efecto puede hacerse esa llamada y va a funcionar, puede inducir a confusión a quien lea el código. Si se observa una llamada un método a partir de un objeto lo normal es pensar que se está llamando a un método de ese objeto para que realice operaciones teniendo en cuenta los atributos de ese objeto. Sin embargo en el caso de un método estático no es así, pues un método estático jamás podrá acceder ni alterar el estado de un objeto.

Es muy importante que *cualquier método estático se invoque siempre referenciado por el nombre de la clase a la que pertenece*, justamente para hacer explícito que se trata de un método estático sin más que ver la sentencia donde se invoca, quedando claro que hace algo para la clase, y no para un objeto particular. Es decir, algo como:

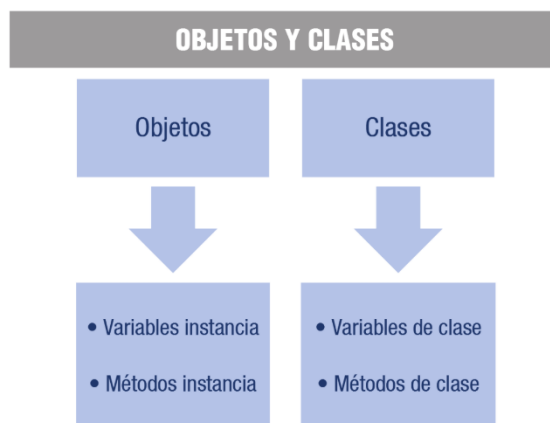
```
NombreClase.nombreMetodoEstatico()
```

En la **biblioteca de Clases de la API Java** existen muchas clases que contienen métodos estáticos, ya los iremos viendo poco a poco según los vayamos necesitando.

Un ejemplo típico de clase con muchos métodos estáticos es la clase **Math** que se encuentra en el paquete **java.lang** (clase **java.lang.Math**).

Esta clase tiene además la particularidad de que no puede ser instanciada, es decir, que no se pueden tener objetos de ella. Por tanto todos sus métodos son obligatoriamente estáticos.

Todos los métodos de la clase **Math** son estáticos.



TEMA 3: UTILIZACIÓN DE OBJETOS

Esta clase podemos imaginárla como una especie de "*caja de herramientas*" (*toolbox*) que proporciona una serie de métodos para ser usados sin necesidad de objetos instancia de la clase. De hecho es para lo que se utiliza, para contener una extensa lista de funciones matemáticas a las que podemos invocar cuando nos haga falta.

Aquí tienes algunos ejemplos de métodos de la clase (todos estáticos):

```
static double abs(double a) // Calcula y devuelve el valor absoluto del parámetro
static double cos(double a) // Calcula y devuelve el coseno trigonométrico del parámetro
static double exp(double a) // Calcula y devuelve el número e elevado a la potencia indicada por el parámetro
static double log(double a) // Calcula y devuelve el logaritmo neperiano del parámetro
static double max(double a, double b) // Calcula y devuelve el máximo de los dos parámetros
static double pow(double a, double b) // Calcula y devuelve el valor del primer parámetro elevado al segundo
static double sqrt(double a) // Calcula y devuelve la raíz cuadrada del parámetro
static double random() // Devuelve un valor aleatorio mayor o igual que 0.0 y menor que 1.0
```

Esta clase también proporciona algunos atributos públicos con algunas de las constantes más habituales en matemáticas:

- la constante "e" de Euler (**Math.E**);
- el número pi: π (**Math.PI**).

```
public static double E // El valor aproximado del número e (2,718281828459045)
public static double PI // Un valor aproximado de pi (3.141592653589793)
```

Estos atributos, al igual que los métodos, son también estáticos (tienen existencia y valor independientemente de la existencia o no de objetos instancia de la clase) pues todos los miembros (métodos y atributos) de la clase **Math** son estáticos.

Aquí tienes algunos ejemplos de uso de estos métodos y atributos estáticos:

```
// Funciones trigonométricas
double angulo = 45.0 * Math.PI/270.0;
System.out.println("cos(" + angulo + ")= " + Math.cos(angulo));

//La función exponencial devuelve el número e elevado a una potencia
System.out.println("exp(10.0)= " + Math.exp(10.0));

// Logaritmo natural (de base e) de un número
System.out.println("log(10.0)= " + Math.log(10.0));
System.out.println("log(Math.E)= " + Math.log(Math.E));

// Elevar 10 a la potencia 2.5
System.out.println("pow(10.0, 2.5)= " + Math.pow(10.0,2.5));

// Hallar la raíz cuadrada de un número
System.out.println("Raíz cuadrada de 3= " + Math.sqrt(3));
```

```
cos(0.5235987755982988)= 0.8660254037844387
exp(10.0)= 22026.465794806718
log(10.0)= 2.302585092994046
log(Math.E)= 1.0
pow(10.0, 2.5)= 316.22776601683796
Raíz cuadrada de 3= 1.7320508075688772
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Otro ejemplo de clase con miembros estáticos que ya has utilizado es la clase **System** (de la que luego volveremos a hablar en la unidad), que se encuentra en el paquete **java.lang** y que contiene abundantes miembros estáticos. De hecho, al igual que sucede con **Math**, esta clase tampoco puede ser instanciada, así que todos sus miembros son estáticos.

Veamos un pequeño ejemplo de acceso a algunos miembros estáticos (atributos y métodos) de esta clase:

```
// Hora actual en milisegundos
long horaMiliseg = System.currentTimeMillis() ;
System.out.println("Hora actual en milisegundos: " + horaMiliseg);

// Separador en la ruta de archivos
String separador = System.getProperty("file.separator") ;
System.out.println("Separador de carpetas en las rutas: " + separador);

// Idioma
String idioma = System.getProperty("user.language") ;
System.out.println("Idioma: " + idioma);

// Versión de Java
String versionJava = System.getProperty("java.version") ;
System.out.println("Versión de Java: " + versionJava);
```

Si ejecutáramos esas líneas, la salida del programa nos mostraría algo parecido a esto (dependiendo de las características de vuestro sistema):

```
Hora actual en milisegundos: 1604375707426
Separador de carpetas en las rutas: \
Idioma: es
Versión de Java: 1.8.0_151
```



Para saber más

Puedes consultar la documentación de esta clase y sus métodos en siguiente enlace:

[Documentación de la clase Math de la API de Java](#)

5.6.1. EJEMPLOS: GENERAR NÚMEROS ALEATORIOS.

En Java podemos generar números pseudoaleatorios de dos formas:

- Utilizando el método estático **random** de la clase **Math**.
- Utilizando métodos la clase **Random**.

El método **random** de la clase **Math** devuelve un número aleatorio de tipo **double** mayor o igual que **0.0** y menor que **1.0**.

Para generar un número entre 0 y el número que deseemos, basta con hacer un simple cambio de escala multiplicando por el factor de escala que necesitemos. Por ejemplo para escalar el rango de [0,1[a [0,25[, podríamos escribir: **Math.random()*25**. Ésto generaría un número entre 0 y 25, donde este último no se incluye. El máximo sería 24,99999...

Si lo que queremos generar es un **número entero**, entonces debemos truncar el resultado anterior quedándonos sólo con la parte entera. Para hacerlo podemos por ejemplo realizar un casting o conversión explícita a **int**:

```
(int)(Math.random()*25); // El resultado final es convertido a tipo int
```

En este caso generaremos un número entero aleatorio entre 0 y 25 pero que nunca llegará a ser 25. Otra forma de hacerlo podría ser mediante el uso del método **Math.floor**, aunque como este método devuelve un número real (**double**) también tendríamos que volver a hacer una conversión explícita si lo que necesitamos es un tipo entero:

```
Math.floor((Math.random()*25)); // El resultado final es de tipo double
(int) Math.floor((Math.random()*25)) // El resultado final es convertido a tipo int
```

Aquí tienes un ejemplo del proceso paso a paso:

```
double aleatorioReal;
double aleatorioRealEscalado;
int aleatorioEntero1;
int aleatorioEntero2;

aleatorioReal= Math.random();
aleatorioRealEscalado= aleatorioReal*25;
aleatorioEntero1= (int)aleatorioRealEscalado;
aleatorioEntero2= (int) Math.floor (aleatorioRealEscalado);
System.out.println ("Aleatorio real: " + aleatorioReal);
System.out.println ("Aleatorio real escalado al intervalo [0,25[: " + aleatorioRealEscalado);
System.out.println ("Aleatorio entero con casting (int): " + aleatorioEntero1);
System.out.println ("Aleatorio entero con Math.floor: " + aleatorioEntero2);
```

```
Aleatorio real: 0.7318010261640145
Aleatorio real escalado al intervalo [0,25[: 18.295025654100364
Aleatorio entero con casting (int): 18
Aleatorio entero con Math.floor: 18
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Si quisiéramos también alcanzar el 25, podríamos por ejemplo escalar entre 0 y 26 (25+1):

```
(int)( Math.random() * (25+1) ); // Aleatorio entero entre 0 y 25, ambos incluidos
```

En general, se trataría de sumar 1 al máximo si deseamos que el máximo esté incluido. Por ejemplo, si queremos generar aleatorios en el rango $[0, N]$, es decir entre 0 y N, donde ambos extremos están incluidos, podríamos hacer algo como:

```
(int)( Math.random() * (N+1) ); // Aleatorio entero entre 0 y N, ambos incluidos
```

Otra opción podría ser utilizar el método **round** de la clase **Math**, que nos permite redondear un número, eso haría que:

1. si el número aleatorio generado y escalado (multiplicado por 25) fuera 24,5 o superior (hasta 24,99999...), entonces el resultado final sería 25;
2. el número aleatorio generado y escalado fuera 25,499999... o inferior (hasta 24,0), entonces el resultado final sería 24.

La forma de generar el número quedaría entonces así:

```
Math.round(Math.random() * N); // Aleatorio entero entre 0 y N, ambos incluidos
```

Si al experimento anterior le añadimos esa línea:

```
double aleatorioReal;
double aleatorioRealEscalado;
int aleatorioEntero1;
int aleatorioEntero2;

aleatorioReal= Math.random();
aleatorioRealEscalado= aleatorioReal*25;
aleatorioEntero1= (int)aleatorioRealEscalado;
aleatorioEntero2= (int) Math.floor (aleatorioRealEscalado);
System.out.println ("Aleatorio real: " + aleatorioReal);
System.out.println ("Aleatorio real escalado al intervalo [0,25[: " + aleatorioRealEscalado);
System.out.println ("Aleatorio entero con casting (int): " + aleatorioEntero1);
System.out.println ("Aleatorio entero con Math.floor: " + aleatorioEntero2);
System.out.println ("Aleatorio real escalado y redondeado a entero: " + Math.round(aleatorioRealEscalado));
```

```
Aleatorio real: 0.3908175527635215
Aleatorio real escalado al intervalo [0,25[: 9.770438819088037
Aleatorio entero con casting (int): 9
Aleatorio entero con Math.floor: 9
Aleatorio real escalado y redondeado a entero: 10 --> Pasa a 10
```

```
Aleatorio real: 0.688931852993666
Aleatorio real escalado al intervalo [0,25[: 17.223296324841648
Aleatorio entero con casting (int): 17
Aleatorio entero con Math.floor: 17
Aleatorio real escalado y redondeado a entero: 17 --> Se queda en 17
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Según el número aleatorio generado y escalado este por encima o por debajo del .5.

Otra alternativa para generar números aleatorios fácilmente en Java es mediante el uso de la clase **Random**, que se encuentra en el paquete **java.util**.

Para generar números aleatorios con esta clase se debe crear una instancia de la clase **Random**. El algoritmo de generación trabaja con un valor semilla. Si se usa el constructor sin parámetro, el valor de la semilla se obtiene del valor de tiempo en nanosegundos que maneja la máquina virtual Java, así tenemos dos **constructores** en la clase:

- **Random()**: crea un nuevo generador aleatorio. En cada invocación Java se encarga de que la semilla para generar los números sea siempre distinta.
- **Random(long semilla)**: crea un nuevo generador aleatorio con una semilla especificada.

La clase **Random** brinda cierta flexibilidad o posibilidades adicionales a **Math.random**, como se puede comprobar en la [referencia de esta clase en la Documentación de la API de Java](#), pues dispone de bastantes métodos. Por ejemplo, tenemos disponibles varios métodos para generar números aleatorios de tipo **int**, **long**, **double**, etcétera.

- **nextBoolean()**: devuelve un valor pseudoaleatorio booleano.
- **nextDouble()**: devuelve un valor pseudoaleatorio de tipo **double** entre 0.0 y 1.0.
- **nextFloat()**: el mismo caso que el anterior pero de tipo **float**.
- **nextInt()**: devuelve un pseudoaleatorio de tipo **int** con valores entre 0 y 2^{32} producidos con aproximadamente la misma probabilidad.
- **nextInt(int n)**: devuelve un número pseudoaleatorio de tipo **int** comprendido entre cero y el valor especificado (excluido).

Aquí tienes un ejemplo de uso de algunos de esos métodos:

```
Random objetoRandom= new Random();
System.out.println ("Aleatorio entero con Random: " + objetoRandom.nextInt());
System.out.println ("Otro aleatorio entero con Random: " + objetoRandom.nextInt());
System.out.println ("Aleatorio entero entre 0 y 25 con Random (sin contar el 25): " + objetoRandom.nextInt(25));
```

Cuyo resultado obviamente es impredecible. Un ejemplo de salida podría ser:

```
Aleatorio entero con Random: -72400602
Otro aleatorio entero con Random: 59409490
Aleatorio entero entre 0 y 25 con Random (sin contar el 25): 19
```



Ejercicio Resuelto

Supongamos que queremos obtener un **número entero aleatorio entre el 1 y el 10 incluyendo ambos**.

Hacerlo usando el método estático `random` de la clase `Math`.

Mostrar retroalimentación

Lo primero que debemos hacer es transformar el tamaño de la escala `[0,1[` (tamaño 1) al tamaño la escala que necesitamos. La escala que necesitamos es de 1 a 10, ambos incluidos. Eso significa que tenemos que "ampliar" (multiplicar) el número obtenido a una escala de tamaño $(10-1+1)$ (el máximo menos el mínimo más 1, si queremos que ambos estén incluidos).

Por tanto, a partir del aleatorio "básico" en el rango `[0,1[`:

```
Math.random()
```

habrá que pasar al nuevo rango `[0, 1*10[`:

```
Math.random() * (10-1+1)
```

o simplemente:

```
Math.random() * 10
```

```
(int) (Math.random() * 10)
```

Ahora bien, dado que nosotros queremos el número en el intervalo `[1,10]`, hay que llevar a cabo un cambio de origen, desplazando (sumando) al origen (valor mínimo) la cantidad de desplazamiento (+1):

```
1 + (int) (Math.random() * 10)
```

De este modo obtendremos un número entero en el intervalo `[1,11[` (incluyendo 1 y sin incluir 11) o bien `[1,10]` (incluyendo 1 y 10).

Hacerlo ahora usando la clase `Random`.

Mostrar retroalimentación

Podrías crear un objeto `Random`:

```
Random objetoRandom= new Random();
```

y a continuación usar el método `nextInt` con el parámetro 10:

```
objetoRandom.nextInt(10)
```

Pero comprobarás que te falta algo más. ¿Se te ocurre qué?



Ejercicio Propuesto

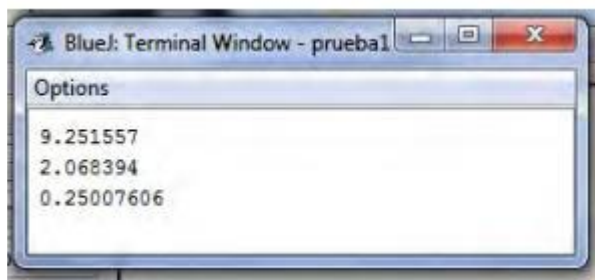
Deseamos generar números enteros aleatorios entre 10 y 20, ambos inclusive. ¿Cómo lo harías?

A continuación, puedes ver algún ejemplo de generación de números aleatorios con la clase Random.

EJERCICIO EJEMPLO DE USO DE NÚMEROS ALEATORIOS EN JAVA

Vamos a resolver ejercicios ilustrativos del uso de números aleatorios en Java. El primero de ellos: crear el código de un programa en el que se declaren tres variables tipo float a, b y c, cuyo valor se muestre en pantalla y deberá estar comprendido entre cero y 10, excluido el diez.

```
/* Ejemplo uso clase Random() - aprenderaprogramar.com */  
  
import java.util.Random;  
public class Programa {  
    public static void main(String arg[]) {  
        float a, b, c;  
  
        Random rnd = new Random();  
  
        a = (rnd.nextFloat() * 10);  
        b = (rnd.nextFloat() * 10);  
        c = (rnd.nextFloat() * 10);  
  
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(c);  
    }  
}
```

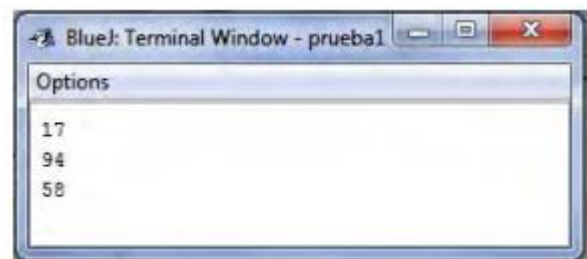


Comentario: la instrucción siguiente (rnd.nextFloat() * 10), se genera un valor de tipo float y dado que la variables a, b, y c han sido declaradas como float al inicio del programa, los tipos son coherentes.

EJERCICIO

Crear el código de un programa en el que se declaren tres variables tipo int a, b y c, cuyo valor se muestra en pantalla y debe estar comprendido entre cero y 100, utilizando el método nextInt de la clase Random.

```
/* Ejemplo uso clase Random() - aprenderaprogramar.com */  
  
import java.util.Random;  
public class Programa {  
    public static void main(String arg[]) {  
        int a, b, c;  
  
        Random rnd = new Random();  
  
        a = rnd.nextInt(101);  
        b = rnd.nextInt(101);  
        c = rnd.nextInt(101);  
  
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(c);  
    }  
}
```



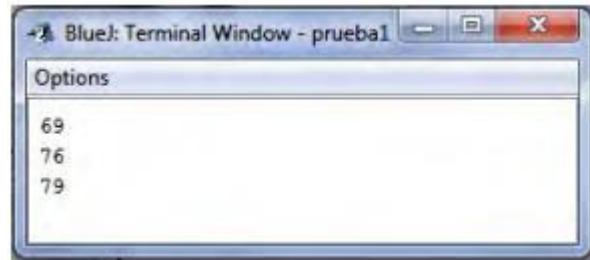
Fíjate que el tipo devuelto por el método nextInt es un int, mientras que el tipo devuelto por el método nextDouble es un double ó el tipo devuelto por el método nextFloat es un float

TEMA 3: UTILIZACIÓN DE OBJETOS

EJERCICIO

Crear el código de un programa que genera tres números enteros aleatorios a, b, c, comprendidos entre 65 y 90, ambos inclusive. Los mostraremos en pantalla.

```
/* Ejemplo uso clase Random() - aprenderaprogramar.com */  
  
import java.util.Random;  
public class Programa {  
    public static void main(String arg[]) {  
        int a, b, c;  
  
        Random rnd = new Random();  
  
        a = (rnd.nextInt(26) + 65);  
        b = (rnd.nextInt(26) + 65);  
        c = (rnd.nextInt(26) + 65);  
  
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(c);  
    }  
}
```



Comentario: Si quieres puedes usar otra fórmula. Puedes escribir con igual resultado, lo siguiente: $a = \text{rnd.nextInt}(90 - 65 + 1) + 65$; $b = \text{rnd.nextInt}(90 - 65 + 1) + 65$; $c = \text{rnd.nextInt}(90 - 65 + 1) + 65$;

5.6.2. MÉTODOS "FÁBRICA" O PSEUDOCONSTRUCTORES.

Hemos visto algunos casos de métodos de una clase que devuelven como resultado objetos instancia de esa misma clase. En algunas ocasiones puede deberse a que se trata de un método que realiza algún tipo de operación entre el objeto cuyo método se invoca y los parámetros que recibe, devolviendo un nuevo objeto resultado de esa operación. Algunos ejemplos de esto podrían ser los métodos **union** o **intersection** de la clase **Rectangle**, que devuelven un nuevo rectángulo como resultado de realizar una unión o intersección con otro rectángulo. Otro ejemplo podrían ser los métodos **concat** o **trim** de la clase **String** que permiten concatenar cadenas o eliminar los espacios en blanco en sus extremos. Son métodos que no modifican el objeto cambiando sus atributos, sino que generan un nuevo objeto con esos cambios.

En otros casos se trata de un **método estático** que genera un nuevo objeto de esa clase a partir de ciertos parámetros, como si fuera un constructor. Hay quien llama a estos métodos "pseudoconstructores" o métodos "fábrica" pues se trata de métodos que "fabrican" o instancian objetos de una manera similar a como lo hacen los constructores.

Existen algunas clases en la API de Java que disponen de métodos de este tipo. Por ejemplo, la clase **LocalTime**, que no tiene constructores públicos sino que proporciona un conjunto de métodos estáticos para poder instanciar un objeto de tipo **LocalTime**. Un poco más adelante, en el apartado dedicado al trabajo con horas y fechas, verás algunos ejemplos de uso de esos métodos "fábrica" que generan nuevos objetos de tipo **LocalDate**, **LocalTime** o **LocalDateTime** como si fueran constructores.

5.7 CLASES ENVOLTORIO EN JAVA.

Como ya has visto anteriormente en el temario, Java usa tipos primitivos, como **int** o **double**, para contener los tipos de datos básicos admitidos por el lenguaje, hacer operaciones con ellos, etc.

Pero hay ocasiones en los que necesitaremos utilizar un dato representado como un objeto en lugar de como tipo primitivo. Por ejemplo, muchas de las estructuras de datos estándar implementadas por Java, y que veremos en temas posteriores, operan con objetos, lo que significa que no se puede usar estas estructuras de datos para almacenar tipos primitivos. Para gestionar estas situaciones Java proporciona **envoltorios** o wrappers de tipo, que son **clases que encapsulan un tipo primitivo** dentro de un objeto.

Cada tipo primitivo en Java tiene su correspondiente clase envoltorio.

Clases envoltorio

Tipo primitivo	Clase envoltorio
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

En principio, los objetos se construyen pasando el valor al constructor correspondiente. Por ejemplo, dado el dato primitivo siguiente:

```
int entero = 20;
```

construiríamos un objeto a partir de él:

```
Integer oEntero = new Integer(entero); //de primitivo a objeto se llama boxing.
int miEntero = oEntero.intValue();    //de objeto a primitivo se llama unboxing.
```

Pero estas operaciones complican excesivamente el código. Así que a partir de J2SE 5.0 se introdujo una conversión automática (**autoboxing**) que permite asignar y obtener los tipos primitivos sin necesidad de utilizar las clases envoltorio.

Autoboxing es la conversión automática que el compilador de Java hace que entre los tipos primitivos y sus clases de objetos. Por ejemplo, la conversión de un **int** a un **Integer**, un **double** con un **Double**, y así sucesivamente. Si la conversión es de objeto a dato primitivo entonces se denomina **Unboxing**.

De este modo, podríamos hacer si problema:

```
int entero = 20;
Integer oEntero = entero;
```

O bien simplemente:

```
Integer oEntero = 20;
```

Y el compilador hará automáticamente la conversión.

Para saber más

Es importante que mires detenidamente el siguiente enlace, donde se explica con ejemplos conceptos sobre este asunto. [Autoboxing y Unboxing con ejemplos](#)

Por último, debes tener en cuenta que las **clases envoltorio** son inmutables (como ya vimos en el caso de la clase **String**). Es decir, que una vez que instanciamos un objeto de estas clases y le damos un valor ya no podrá cambiar su estado. Eso significa que cualquier operación que se realice sobre estos objetos dará lugar como resultado a una nueva instancia, pues el objeto original nunca se modificará.

Por ejemplo, dado:

```
Integer oNumero = 3 ;  
oNumero += 3 ;
```

Se crea un objeto **oNumero** de clase entero con valor 3, y después de ejecutarse **oNumero +=3**, se crea una nueva instancia para guardar el valor 6 y la primera instancia que se creó se pierde, quedándose **oNumero** apuntando a una nueva instancia de la clase **Integer** que almacena el valor 6.

Si quisiéramos guardar ambas instancias, necesitaríamos dos variables de tipo referencia para almacenar una referencia a cada objeto creado:

```
Integer oNumero1 = 3 ;  
Integer oNumero2;  
oNumero2 = oNumero1 + 3 ;
```

5.8 REPRESENTACIÓN TEXTUAL DE UN OBJETO EN JAVA: MÉTODO **toString**.

En Java, toda clase, por el hecho de ser una clase Java, dispone del método **toString**.

El propósito de este método es **asociar a todo objeto un texto representativo de su contenido**. Es decir, que se trata de un método que devuelve una representación "textual" del contenido del objeto. Ahora bien, ¿qué formato tendrá esa representación textual para cada clase que se escriba en Java? Esa es una cuestión que habrán decidido los desarrolladores de cada clase en su momento. Si queremos saber qué aspecto tiene la representación textual de los objetos de una clase podemos:

1. consultar la documentación de la clase y observar qué indica acerca de su método **Rectangle**;
2. hacer un pequeño programa de prueba donde creemos un objeto instancia de esa clase y mostremos por pantalla la cadena devuelta por el método **toString** de ese objeto.

Podemos realizar ambos experimentos con un objeto de la clase **Rectangle** que llevamos usando para nuestras pruebas a lo largo de toda la unidad. Si consultamos la [documentación de la API de Java respecto a la clase **Rectangle**](#), podemos observar la siguiente descripción de su método **toString**:

```
toString  
  
public String toString()  
Returns a String representing this Rectangle and its values.  
Overrides:  
toString in class Object  
Returns:  
a String representing this Rectangle object's coordinate and size values.
```

En esa descripción nos indica que el método devuelve una cadena (objeto **String**) con las coordenadas y tamaño del objeto **Rectangle**. La verdad es que no es mucha información pues es lo que podíamos imaginar que haría el método sin necesidad de leer esa descripción.

Si queremos saber algo más sobre el aspecto que tendrá esa cadena de caracteres, podemos optar por hacer un pequeño programa de prueba donde instanciamos un objeto rectángulo y mostremos por pantalla la cadena devuelta por el método **toString**:

TEMA 3: UTILIZACIÓN DE OBJETOS

```
// Creamos un par de objetos Rectangle y almacenamos sus referencias en sendas variables
// de tipo referencia a Rectangle
Rectangle r1= new Rectangle (1,2, 5, 4);
Rectangle r2= new Rectangle (5,3, 4, 6);
// Mostramos por pantalla la "representación textual" ofrecida por el método toString
System.out.println( "r1= " + r1.toString() );
System.out.println( "r2= " + r2.toString() );
```

```
r1= java.awt.Rectangle[x=1,y=2,width=5,height=4]
r2= java.awt.Rectangle[x=5,y=3,width=4,height=6]
```

Podemos observar que el formato de la cadena tiene el siguiente aspecto:

```
java.awt.Rectangle[x=<valorX>,y=<valorY>,width=<valorBase>,height=<valorAltura>]
```

donde **<valorX>**, **<valorY>**, **<valorBase>**, y **<valorAltura>** son los valores de los atributos del objeto rectángulo para cada caso concreto.

¿Para qué me puede servir esta representación textual? Pues por ejemplo para evitarnos tener que acceder explícitamente los valores de todos los atributos cada vez que queramos mostrar el estado del objeto por pantalla. Es decir, en lugar de hacer como hemos estado haciendo hasta el momento:

```
// Mostramos el estado del objeto rectángulo al que apunta r1
System.out.println ("Rectángulo rec1:");
System.out.println ("Ubicación: x=" + rec1.x + " y=" + rec1.y);
System.out.println ("Dimensiones: base= " + rec1.width+ " altura= " + rec1.height);
```

Ahora podemos hacer simplemente:

```
// Mostramos el estado del objeto rectángulo al que apunta r1
System.out.println( "r1= " + r1.toString() );
```

Es posible que en algunos casos te encuentres con clases que no tengan implementado explícitamente el método **toString**. En tal caso, como ya hemos dicho, toda clase Java (en realidad cualquier elemento que sea de tipo "referencia") ya dispone de ese método implícitamente. Pero si el programador no lo ha reescrito, el resultado textual que devuelve es muy "pobre" y tendrá un aspecto parecido a algo como **<nombreClase>@<numero>** (por ejemplo **java.awt.Rectangle@1af7ab3**). Esa información devuelta es conocida como "*hash*" y está relacionada con la posición de memoria en la que se almacena el objeto. Ésa es la famosa "referencia" (número) que se almacena en las variables de tipo referencia y a partir de la cual se consulta en memoria para obtener la información sobre el estado del objeto al que apunta (valor de sus atributos).

Por último, es importante que sepas que para obtener la representación textual (salida del método **toString**) de un objeto no es necesario llamar explícitamente al método **toString** si por ejemplo lo vas utilizar para mostrarlo en pantalla o concatenarlo con otra cadena

Por ejemplo, en los casos anteriores podrías haber escrito directamente **System.out.println ("r1= " + r1)**. Es decir, que **si vamos a interpretar una variable referencia (objeto) como un String, no es necesario invocar explícitamente al método toString** pues Java lo hará implícitamente por nosotros. En tal caso, el código para mostrar el estado de un objeto por pantalla quedaría más simple todavía:

```
// Mostramos el estado del objeto rectángulo al que apunta r1
System.out.println( "r1= " + r1 ); // Llamada "implícita" a r1.toString()
```

6. LIBRERÍA 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.

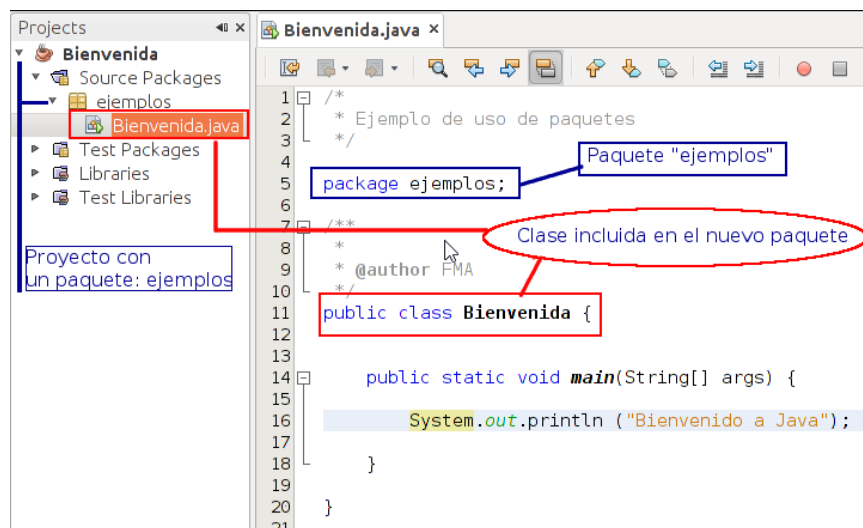
Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete, una especie de "relación de confianza". Es por ello por lo que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.

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.

En el siguiente fichero puedes ver una demostración de cómo hemos creado el proyecto [Bienvenida](#).

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 con una sentencia **import**. Por ejemplo, para importar todas las clases del paquete **java.awt**, se puede hacer así:

```
import java.awt.*;
```

Es importante aclarar que usando el asterisco, importamos las clases del paquete, pero NO se importan las clases de los subpaquetes que pudiera tener.

Las sentencias **import** (puede haber varias) deben aparecer al principio de la clase, justo después de la sentencia **package**, si ésta existiese.

También podemos utilizar una clase sin importarla con la 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);
```

La sentencia **import** admite una variante, conocida como **import static**, la cual permite el uso de métodos y atributos estáticos de otras clases directamente, sin tener que especificar el nombre de la clase importada en nuestro código.

```
import static java.lang.Math.round;
```

En el **import** anterior **round** es un método de la conocida clase **Math**. Esto significa que podremos usarlo en nuestro código directamente, veamos un ejemplo:

```
public static void main(String[] args)
{
    System.out.println(round(20.2));
}
```

El código anterior mostraría **20** por pantalla.



Reflexiona

En unidades anteriores hemos hecho uso de clases como `String`, o `Math`, que si consultamos en la documentación de la API, pertenecen al paquete `java.lang` y sin embargo, nunca hemos tenido que usar ninguna sentencia `import java.lang.String`; ni `import java.lang.Math`; ni tampoco `import java.lang.*`; y a pesar de ello todo funcionaba correctamente.

¿No habíamos quedado en que siempre había que importar las clases de otros paquetes que quisiéramos usar?

Por otra parte, ¿qué es preferible, importar las clases una a una o importar todas las clases del paquete usando el asterisco?

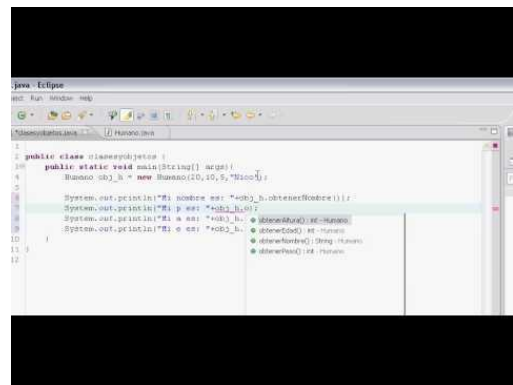
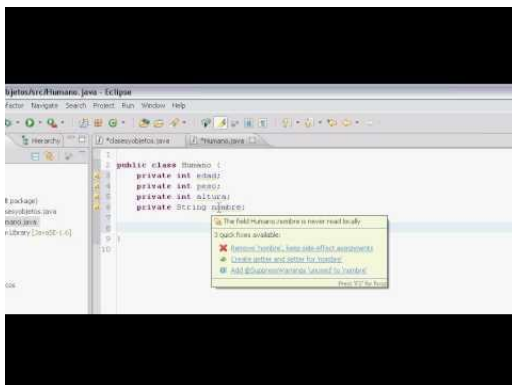
Mostrar retroalimentación

En principio es así, cuando se quiere usar una clase de otro paquete, hay que indicarlo expresamente con una sentencia `import`, pero el paquete `java.lang` es una excepción. Este paquete contiene una serie de clases tan importantes, que sin él sería imposible hacer nada en Java, ya que contiene, por ejemplo, la propia clase `Object`, de la que heredan todas las demás clases, y también toda una serie de clases de uso tan frecuente, que sería realmente tedioso y molesto tener que ir importándolas cada vez que se van a usar, lo que supondría importarlas prácticamente siempre, como es el caso de por ejemplo, `String`. Por eso Java nos ahorra el trabajo e importa implícitamente todas las clases de ese paquete, lo que nos permite usarlas sin preocuparnos de hacer nada más.

Respecto a la segunda cuestión, vamos a responderla con una especie de FAQ sobre el uso de `import`, que pueden ayudar a entender mejor su funcionamiento.

- ✓ **¿Importar las clases hace que mi archivo (.class o .jar) sea más largo?** No, la sentencia `import` sólo proporciona al compilador una ruta donde ir a buscar las clases cuando las necesita.
- ✓ **¿Es menos eficiente importar todas las clases de un paquete usando `*` que importar sólo las que vaya a usar con una sentencia `import` para cada una?** No. La búsqueda de nombres es muy eficiente y no hay diferencia práctica, dado que "no se importa nada realmente", sólo se indica la ruta donde buscar, y si es la misma para toda una serie de clases, es suficiente con poner la ruta para todas, en lugar de poner la ruta una a una.
- ✓ **¿No queda mejor documentado el código si se usan `import` distintos para importar cada clase de forma individual?** Podría parecer que sí, pero la realidad es que NO, porque tiene una serie de desventajas:
 - ✦ Es "duro" acordarse de borrar las clases de la lista cuando dejen de usarse en futuras versiones de un programa, por lo que puede darse el caso de que una clase que ha dejado de usarse siga teniendo su sentencia `import`, por lo que en este caso, al verlo, si vamos a buscar qué se hace con esa clase que ha dejado de usarse nos podemos volver locos, así que en vez de una ayuda a la documentación puede acabar siendo un error de documentación.
 - ✦ Importar explícitamente una única clase permite que "accidentalmente" pueda definir clases propias cuyos nombres entran en conflicto con los nombres de las clases en las librerías estándar, lo cual es poco deseable. Usando un `import` para todo un paquete con `*`, ayudamos en realidad a prevenir ese accidente.
- ✓ **Si ya he importado, por ejemplo, `java.awt.*`, ¿por qué tengo que importar también `java.awt.event.*`?** La sentencia `import` con la plantilla `*` sólo establece una ruta, que sólo permite importar las clases de esa ruta (ese paquete) y no los subpaquetes, que serían rutas distintas. (Piensa que es como si cada paquete fuera una carpeta, que puede tener subcarpetas).
- ✓ **¿Por qué no necesito usar un `import` con las clases `String`, `System`, `Math`, etc.?** Esto ya se explicó con más detalle al comienzo, pero resumidamente, esas clases están en el paquete `java.lang`, que contiene clases tan importantes y necesarias que el lenguaje las incluye por defecto, sin necesidad de usar explícitamente `import`.
- ✓ **¿Es importante el orden en que se escriban las sentencias `import`?** No lo es. Sólo es aconsejable intentar escribirlas de forma que sean más legibles. Por ejemplo, agrupando los `import` que tengan cierta relación, etc.

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 y otro habla sobre la utilización de objetos y clases en Java:



6.2 COMPILAR Y EJECUTAR CLASES CON PAQUETES.

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.

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

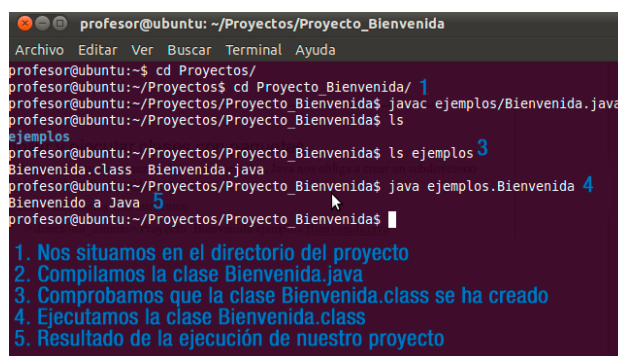
```
/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/Bienvenida.java
```

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**.



```
profesor@ubuntu: ~/Proyectos/Proyecto_Bienvenida
Archivo Editar Ver Buscar Terminal Ayuda
profesor@ubuntu:~$ cd Proyectos/
profesor@ubuntu:~/Proyectos$ cd Proyecto_Bienvenida/
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ javac ejemplos/Bienvenida.java
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ ls
ejemplos
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ ls ejemplos
Bienvenida.class Bienvenida.java
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ java ejemplos.Bienvenida
Bienvenido a Java
```

1. Nos situamos en el directorio del proyecto
2. Compilamos la clase **Bienvenida.java**
3. Comprobamos que la clase **Bienvenida.class** se ha creado
4. Ejecutamos la clase **Bienvenida.class**
5. Resultado de la ejecución de nuestro proyecto

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.

Afortunadamente, si el proyecto lo hemos creado desde un entorno integrado como Netbeans, podemos delegar en que será el entorno quien se encargue de estos detalles, la compilación y ejecución se realizará al ejecutar la opción **Run File** (Ejecutar archivo) o hacer clic sobre el botón **Ejecutar** de la barra de herramientas.

6.3 JERARQUIA DE PAQUETES.

Para organizar mejor las cosas, un **paquete** también puede contener otros paquetes. Es decir, podemos hacer **subpaquetes** de los paquetes y subpaquetes de los subpaquetes y así sucesivamente. Piensa que internamente Java asocia una carpeta a cada paquete, así que podemos tener cualquier jerarquía que paquetes, de la misma forma que podemos tener cualquier jerarquía de carpetas. 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.

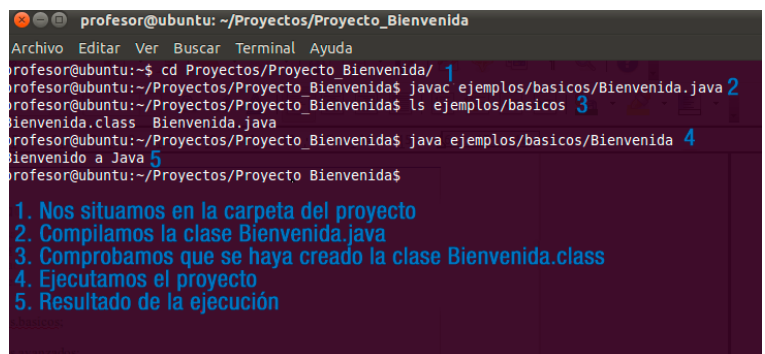
Para compilar, en el directorio del proyecto habría que compilar poniendo todo el **path** hasta llegar a la clase. Es decir, el nombre de la clase va con todos los paquetes separados por puntos, por ejemplo **ejemplos.basicos.Bienvenida**.

La estructura de directorios en el sistema operativo cuando usamos subpaquetes sería:

```
/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/basicos/Bienvenida.java
```

Y la compilación y ejecución sería:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida
$ javac ejemplos/basicos/Bienvenida.java
$ java ejemplos/basicos/Bienvenida
Hola Mundo
```



```
profesor@ubuntu: ~/Proyectos/Proyecto_Bienvenida
Archivo Editar Ver Buscar Terminal Ayuda
profesor@ubuntu:~$ cd Proyectos/Proyecto_Bienvenida/ 1
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ javac ejemplos/basicos/Bienvenida.java 2
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ ls ejemplos/basicos 3
Bienvenida.class Bienvenida.java
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ java ejemplos/basicos/Bienvenida 4
Bienvenido a Java 5
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$
```

1. Nos situamos en la carpeta del proyecto
2. Compilamos la clase Bienvenida.java
3. Comprobamos que se haya creado la clase Bienvenida.class
4. Ejecutamos el proyecto
5. Resultado de la ejecución

La Biblioteca de Clases de Java se organiza haciendo uso de esta jerarquía de paquetes. Así por ejemplo, si quiero acceder a la clase **LocalDate**, tendré que importarla indicando su ruta completa, o sea, **java.time.LocalDate**, así:

```
import java.time.LocalDate;
```

6.4 LIBRERIAS 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, como **Integer**, **Long**, **Float**, **Double**, etc., lo que proporciona una serie de métodos para cada tipo de dato de utilidad, como por ejemplo métodos para 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.
- **java.math.** Contiene herramientas para trabajar con datos numéricos de mayor precisión que la proporcionada por los tipos primitivos tanto para enteros (clase **BigInteger**) como para reales (clase **BigDecimal**).
- **java.time.** Proporciona clases para el trabajo con el tiempo como por ejemplo la clase **LocalDate**, para el tratamiento de fechas o la clase **LocalTime** para el tratamiento de horas.
- **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. Tanto este paquete como el anterior (**java.awt**) pueden considerarse hoy día desfasados y están siendo sustituidos por la tecnología **JavaFX**, aunque aún podrás ver muchas aplicaciones Java con **Swing** y **AWT**.
- **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 que se utiliza para mostrar mensajes de error. Una práctica común en informática es enviar los mensajes de error a un dispositivo de salida diferente al de los mensajes normales. Los mensajes normales se envían a lo que se llama "salida estándar". Los mensajes de error se envían a "error estándar".

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");
```

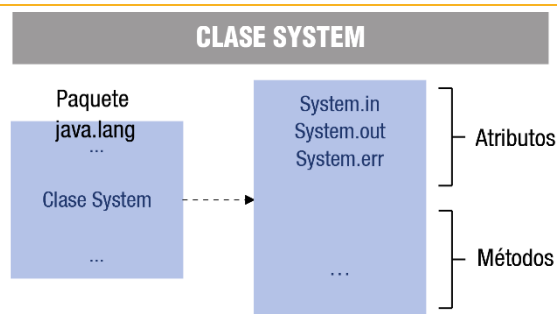
En el siguiente enlace puedes consultar los atributos y métodos de la clase **System** del paquete **java.lang** perteneciente a la Biblioteca de Clases de Java: [Clase System de Java](#).

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 de 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.



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.

TEMA 3: UTILIZACIÓN DE OBJETOS

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.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"**.

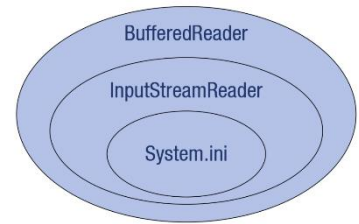
Sin embargo, seguimos necesitando hacer la conversión si queremos leer números. Por ejemplo, si escribimos un entero 32, en cadena obtendremos "32". Si recordamos, para convertir cadenas de texto a enteros se utiliza el método estático **parseInt()** de la clase **Integer**, con lo cual la lectura la haríamos así:

```
int numero = Integer.parseInt(br.readLine());
```

Esa operación de conversión de cadena a número sólo funcionará si la cadena introducida realmente se puede convertir en número, porque representaba a un literal correcto para el tipo numérico de que se trata, en el caso de la sentencia anterior sería **int**. En caso contrario, se produciría una excepción de tipo **NumberFormatException**, (Excepción de Formato de número) que debe ser capturada y tratada convenientemente.

ENCAPSULADO DE OBJETOS

Encapsulado de objetos para entrada por teclado



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.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

/*
 * Ejemplo de entrada por teclado con la clase System
 * @author FMA
 */

public class EntradaTeclado {

    public static void main(String[] args) {
        try
        {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);

            System.out.print("Introduce el texto: ");
            String cad = br.readLine();

            //salida por pantalla del texto introducido
            System.out.println(cad);

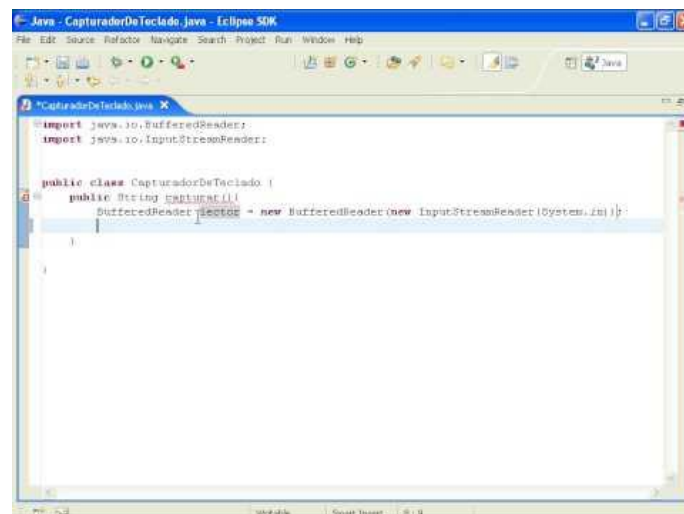
            System.out.print("Introduce un número: ");
            int num = Integer.parseInt(br.readLine());

            // salida por pantalla del número introducido
            System.out.println(num);

        } catch (Exception e) {
            // System.out.println("Error al leer datos");
            e.printStackTrace();
        }
    }
}
```

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 excepción seguramente el programa se pare y no siga ejecutándose. El control de excepciones lo vemos más detenidamente en el siguiente apartado de la unidad.

Te proponemos el siguiente enlace con un descriptivo vídeo sobre cómo capturar datos desde el teclado:



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 dato de tipo 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:

```
import java.util.Scanner;
/*
 * Ejemplo de entrada de teclado con la clase Scanner
 */

/**
 *
 * @author FMA
 */
public class EntradaTecladoScanner {

    public static void main(String[] args) {

        // Creamos objeto teclado
        Scanner teclado = new Scanner(System.in);

        // Declaramos variables a utilizar
        String nombre;
        int edad;
        boolean estudias;
        float salario;

        // Entrada de datos
        System.out.println("Nombre: ");
        nombre=teclado.nextLine();
        System.out.println("Edad: ");
        edad=teclado.nextInt();
        System.out.println("Estudias: ");
        estudias=teclado.nextBoolean();
        System.out.println("Salario: ");
        salario=teclado.nextFloat();

        // Salida de datos
        System.out.println("Bienvenido: " + nombre);
        System.out.println("Tienes: " + edad + " años");
        System.out.println("Estudias: " + estudias);
        System.out.println("Tu salario es: " + salario + " euros");
    }
}
```

Si quieres conocer algo más sobre la clase **Scanner** puedes consultar el siguiente enlace: [Capturar datos desde teclado con Scanner](#).



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()** consideran como cadenas de texto sin formato a todas las variables que escriben; 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. Para ello se utiliza la orden **printf()**.

La orden **printf()** utiliza unos códigos de conversión para indicar de qué tipo es el contenido a mostrar. 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", 12345.1684);
```

Esta orden mostraría el número **12.345,17** por pantalla.

Es importante aclarar que estos códigos de conversión formatean la salida, mostrándola de una manera o de otra, pero no cambian el valor que internamente tengan las variables. Si en el caso anterior, en lugar de imprimir un valor concreto, **12345.1684**, se hubiera impreso una variable con ese mismo valor, aunque se mostrara su valor como **12.345,17** por pantalla, la variable seguiría teniendo almacenado el valor **12345.1684**

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 conocer algo más sobre la orden **printf()**, en el siguiente enlace tienes varios ejemplos de utilización:

[Salida de datos con la orden printf\(\)](#) Y [Entrada y salida de datos](#).

En este otro enlace, puedes ver un ejemplo interesante de uso de la clase **Scanner** junto con las órdenes **printf()** y **println()**. Para ello, descárgate el fichero, descomprímelo y haz doble clic en el fichero .html

[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.

La separación o diferencia entre la salida estándar y la de error se puede utilizar para redirigir los mensajes de error a un lugar diferente que los mensajes normales.

Por defecto, tanto **out** como **err** tienen su destino en 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.

Aunque por defecto vemos la salida de error en la consola, es posible redirigirla a un fichero, por ejemplo, cómo podemos ver en el siguiente para saber más.

Más información sobre la entrada, salida y error estándar. [Flujos de datos estándar](#).

8. MANIPULACIÓN DE LA FECHA Y LA HORA EN JAVA.

Java tiene una API de fechas desde el JDK 1.1., pero su diseño no fue muy afortunado debido a que se demostró que tenía algún que otro problema de seguridad y lo más importante, que su uso no era intuitivo. Así, por ejemplo, el mes comienza desde **0**, cuando intuitivamente el usuario de esta clase tendería a pensar intuitivamente que comienza en **1**. etc.

Posteriormente se intentó rectificar con la introducción de la API Calendar en el JDK 1.4 pero también hubo problemas similares.

Y finalmente, con el JDK 8 surgió una nueva API de fecha y hora moderna y relativamente fácil de usar.

El paquete **java.time** contiene las clases relativas al tiempo (**LocalDate**, **LocalTime** y **LocalDateTime**).

El paquete **java.time.format** contiene la clase **DateTimeFormatter**, por si necesitamos dar formato a las fechas. También incluye como el uso de zonas horarios con **ZonedDateTime** y **Period** para determinar el periodo entre dos fechas y **Duration** para determinar la duración entre dos horas.

8.1 LocalDate

Las clases relativas al tiempo en Java utilizan métodos estáticos, que devuelven una referencia del tipo de la clase que se ha utilizado para ejecutarlos. Si queremos trabajar con fechas podemos usar la clase **LocalDate**. Veamos un ejemplo:

```
package fechas;

import java.time.LocalDate;

/**
 * Impresión de la fecha
 * @author JJBH
 */
public class Fechas {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        LocalDate ahora = LocalDate.now();
        System.out.println(ahora);
    }
}
```

El resultado de ejecutar el programa puede ser algo como:

```
run:
2019-09-05
BUILD SUCCESSFUL (total time: 0 seconds)
```

Como ves, la impresión por consola de la fecha nos muestra primero el año, luego el mes y después el día, siguiendo la norma ISO 8601. Más abajo hay un enlace a un artículo donde puedes ampliar información sobre esta norma.

TEMA 3: UTILIZACIÓN DE OBJETOS

Puedes crear un **LocalDate** mediante el uso del método **of(int year, int month, int dayOfMonth)**. Por ejemplo:

```
LocalDate fecha = LocalDate.of(1945, 11, 30) ;
```

crea una fecha correspondiente al 30 de noviembre de 1945. También se puede hacer uso del enumerado **Month** para dar legibilidad al código, de tal modo que ese mismo ejemplo podría hacerse así:

```
LocalDate fecha = LocalDate.of(1945, Month.NOVEMBER, 30) ;
```

También se puede emplear el método **parse** para crear un objeto de tipo **LocalDate** a partir de una cadena de texto. Este método admite dos argumentos:

- El primero es la cadena de texto que quieres transformar en un objeto del tipo temporal que quieras.
- El segundo es el objeto de tipo **DateTimeFormatter** con el que le vas a decir al método **parse** cómo aparece la fecha expresada en la cadena de texto, para que la entienda.

Podemos ver un breve ejemplo:

```
import java.time.LocalDate;

/**
 * Ejemplo parse
 * @author Profesor
 */
public class EjemploParse {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Crea un objeto LocalDate
        LocalDate lt = LocalDate.parse("2019-12-26");

        // Escribe el resultado
        System.out.println("LocalDate : " + lt);
    }
}
```

que dará como resultado:

```
run:
LocalDate : 2019-12-26
BUILD SUCCESSFUL (total time: 0 seconds)
```

El el siguiente artículo puedes aprender más sobre la norma ISO 8601 [Norma ISO 8601](#)

En el siguiente enlace puedes consultar la API de la clase **LocalDate**. [API de la clase LocalDate](#)

8.2 LocalDateTime

Similar a la clase **LocalDate**, cuando necesitamos operar con horas podremos utilizar la clase **LocalTime**. Veamos un ejemplo:

```
import java.time.LocalTime;

/**
 * Ejemplo de uso de LocalTime
 * @author Profe
 */
public class Horas {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // El método of devuelve una nueva instancia de LocalTime con la hora,
        // minutos, segundos y nanosegundos.
        // Se lanza una excepción DateTimeException si se proporciona algún
        // parámetro no válido
        LocalTime hora = LocalTime.of(20, 30, 45, 35);

        System.out.println(hora.toString());

        // Devuelve la instancia de la hora local del reloj del sistema
        System.out.println(LocalTime.now());
    }
}
```

El resultado de ejecutar el programa puede ser algo como:

```
run:
20:30:45.000000035
14:55:50.920
BUILD SUCCESSFUL (total time: 0 seconds)
```

que como vemos nos imprime la hora indicada con el método **of**, y a continuación la hora actual mediante el método **now()**.

Al igual que en el caso de **LocalDate**, se puede emplear el método **parse** para crear un objeto de tipo **LocalTime** a partir de una cadena de texto.

Por ejemplo:

```
import java.time.LocalTime;

/**
 * Ejemplo parse
 * @author Profesor
 */
public class EjemploParseT {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Crea un objeto LocalTime
        LocalTime localt = LocalTime.parse("10:25:45");

        // Escribe el resultado
        System.out.println("LocalTime : " + localt);
    }
}
```

que dará como resultado:

```
run:
LocalTime : 10:25:45BUILD SUCCESSFUL (total time: 0 seconds)
```

Es muy recomendable que veas los ejemplos que se encuentran en la siguiente página: [Ejemplos con LocalTime \(en inglés\)](#)

En el siguiente enlace puedes consultar la API de la clase **LocalTime**. [API de la clase LocalTime](#)

8.3 LocalDateTime

Esta clase se usa para representar la fecha (año, mes, día) junto con la hora (hora, minuto, segundo, nanosegundo) y es la combinación de **LocalDate** y **LocalTime**.

En el siguiente ejemplo podemos observar el uso de la clase mediante ejemplos:

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

/**
 * Clase de ejemplo para mostrar algunos métodos de la clase LocalDateTime
 * @author JJBH
 */
public class FechaYHora {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Construir un LocalDateTime a partir de un LocalDate y un LocalTime
        LocalDate fecha = LocalDate.of(1989, 10, 21);
        LocalTime hora = LocalTime.now();
        LocalDateTime dateTime = LocalDateTime.of(fecha, hora);
        // Y escribirlo por consola
        System.out.printf("La hora es: %s\n", dateTime.toString());

        // Probamos a sumar y restar.
        LocalDateTime ahora = LocalDateTime.now();
        System.out.printf("La hora es: %s\n", ahora);
        System.out.printf("Hace seis meses sería %s\n", LocalDateTime.now().minusMonths(6));
        System.out.printf("La hora más 20 horas más sería: %s\n", ahora.plusHours(20));
        System.out.printf("Y hace dos días: %s\n", ahora.minusDays(2));
    }
}
```

TEMA 3: UTILIZACIÓN DE OBJETOS

y el resultado sería:

```
run:
La hora es: 1989-10-21T17:36:29.037
La hora es: 2019-09-09T17:36:29.080
Hace seis meses sería 2019-03-09T17:36:29.081
La hora más 20 horas más sería: 2019-09-10T13:36:29.080
Y hace dos días: 2019-09-07T17:36:29.080
BUILD SUCCESSFUL (total time: 0 seconds)
```

Como puedes ver, hay muchos métodos que nos permiten sumar o restar horas, meses, etc.

Puedes descargar el proyecto a continuación: [FechaYhora](#)

Al igual que en los casos anteriores, en este caso también podemos usar el método **parse** para crear un objeto de tipo **LocalDateTime** a partir de una cadena de texto.

Por ejemplo:

```
import java.time.LocalDateTime;

/**
 * Ejemplo parse
 * @author Profesor
 */
public class EjemploParseDT {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Crea un objeto LocalDateTime
        LocalDateTime ltt = LocalDateTime.parse("2020-11-28T19:34:50.63");

        // Escribe resultado
        System.out.println("LocalDateTime : " + ltt);
    }
}
```

que nos ofrecerá como resultado:

```
run:
LocalDateTime : 2020-11-28T19:34:50.630
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tienes información adicional y claros ejemplos para operar con fechas y horas en el siguiente artículo. En particular fíjate en la breve pero clara explicación sobre **ZonedDateTime**, **Period** y **Duration**: [Ejemplos con fechas](#)

En el siguiente enlace puedes consultar la API de la clase **LocalDateTime**. [API de la clase LocalDateTime](#)

8.4 Formateado de fechas.

Nos puede interesar dar formato a las fechas, y en ese caso usaremos la clase **DateTimeFormatter**.

Comentamos anteriormente que la norma ISO 8601 establece el formato predeterminado en las fechas y horas. Esto significa que, al imprimirlas vamos a obtener una fecha u hora con este formato.

Hay que tener en cuenta lo siguiente, hay unas constantes que podemos usar con la clase y que son:

- **ISO_LOCAL_DATE** puede utilizarse con **LocalDate** y **LocalDateTime** (contienen una fecha).
- **ISO_LOCAL_TIME** puede utilizarse con **LocalTime** y **LocalDateTime** (contienen una hora).
- **ISO_LOCAL_DATE_TIME** puede utilizarse solo con **LocalDateTime** (contiene una fecha y una hora).

Valiéndonos de un ejemplo, veamos algunas cosas:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

/**
 * Probando el formateo en el tiempo, con Java
 * @author jjber
 */
public class Formateando {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        LocalDateTime fecha = LocalDateTime.now();

        DateTimeFormatter isoFecha = DateTimeFormatter.ISO_LOCAL_DATE;
        System.out.println("Sin formato: " + fecha);
        System.out.println("Con formato: " + fecha.format(isoFecha));

        DateTimeFormatter isoHora = DateTimeFormatter.ISO_LOCAL_TIME;
        System.out.println(fecha.format(isoHora));

        LocalTime hora = LocalTime.now();
        DateTimeFormatter f = DateTimeFormatter.ofPattern("'Son las' h 'y' mm");
        //System.out.println(hora);
        System.out.println(hora.format(f));

        LocalDateTime otraHora = LocalDateTime.now();
        DateTimeFormatter formato = DateTimeFormatter.ofPattern("hh:MM");
        System.out.println("Hora:mes = " + otraHora.format(formato));
        DateTimeFormatter formato2 = DateTimeFormatter.ofPattern("hh:mm");
        System.out.println("Hora:minutos = " + otraHora.format(formato2));
    }
}
```

cuyo resultado por consola da:

```
run:
Sin formato: 2019-09-09T17:58:10.454
Con formato: 2019-09-09
17:58:10.454
Son las 5 y 58
Hora:mes = 05:09
Hora:minutos = 05:58
BUILD SUCCESSFUL (total time: 0 seconds)
```

TEMA 3: UTILIZACIÓN DE OBJETOS

Fíjate en el uso del método **ofPattern** que acepta un **String** y permite darle cualquier formato a un objeto de tipo **LocalDate**, **LocalTime** o **LocalDateTime** (date cuenta cómo si usamos m en minúsculas se refiere a minutos y si es en mayúsculas se refiere a mes). En la cadena de texto que utilizamos como patrón puede haber letras que simbolizan un elemento temporal. La lista completa podemos verla en [API de la clase](#) o texto entre comillas simples, que aparece tal cual al imprimir.

El método **parse** se utiliza para crear un objeto de tipo **LocalDate**, **LocalTime** o **LocalDateTime** a partir de una cadena de texto. Este método admite dos argumentos:

- El primero es la cadena de texto que quieres transformar en un objeto del tipo temporal que quieras.
- El segundo es el objeto de tipo **DateTimeFormatter** con el que le vas a decir al método **parse** cómo aparece la fecha expresada en la cadena de texto, para que la entienda.

El primer argumento se ha de corresponder con el formato que se vaya a utilizar para "traducirlo" a una instancia de alguna clase relativa al tiempo en Java. El método **parse** puede aceptar un único argumento si la cadena de texto se corresponde con el formato ISO predeterminado.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/**
 *
 * @author Profesor
 */
public class EjemploParse {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        LocalDateTime hora = LocalDateTime.parse("2019-03-27T18:14:01.184");
        DateTimeFormatter formato = DateTimeFormatter.ofPattern("'Hoy es' d 'del mes' M 'del año' yyyy. 'Y son las' kk 'horas.'");
        System.out.println(hora.format(formato));
    }
}
```

que mostrará por consola:

```
run:
Hoy es 27 del mes 3 del año 2019. Y son las 18 horas.
BUILD SUCCESSFUL (total time: 0 seconds)
```


8.5 ChronoUnit

El enumerado **ChronoUnit** define las unidades utilizadas para medir el tiempo.

Por ejemplo, el método **ChronoUnit.between** es útil cuando queremos medir una cantidad de tiempo en una unidad de tiempo sencilla, como días o segundos. Es decir, trabaja con objetos basados en tiempo pero retorna una unidad simple. En el siguiente código podemos ver ejemplos de uso:

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

/**
 * Ejemplo ChronoUnit
 * @author Profe
 */
public class CronoUnit {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Fecha actual
        LocalDate hoy = LocalDate.now();
        System.out.println("Fecha actual: " + hoy);

        // Añadir una semana a la fecha actual
        LocalDate semanaSig = hoy.plus(1, ChronoUnit.WEEKS);
        System.out.println("Una semana después: " + semanaSig);

        // Añadir un mes a la fecha actual
        LocalDate mesSig = hoy.plus(1, ChronoUnit.MONTHS);
        System.out.println("Un mes después: " + mesSig);

        // Añadir un año
        LocalDate sigYear = hoy.plus(1, ChronoUnit.YEARS);
        System.out.println("Un año después: " + sigYear);

        // Añadir una década
        LocalDate sigDecada = hoy.plus(1, ChronoUnit.DECADES);
        System.out.println("Una década después: " + sigDecada);

    }
}
```

que nos mostrará por consola algo similar a:

```
run:
Fecha actual: 2019-09-13
Una semana después: 2019-09-20
Un mes después: 2019-10-13
Un año después: 2020-09-13
Una década después: 2029-09-13
BUILD SUCCESSFUL (total time: 0 seconds)
```

TEMA 3: UTILIZACIÓN DE OBJETOS

En este ejemplo comentado línea a línea puedes aprender más sobre operaciones con fechas:

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;
import java.time.temporal.ChronoUnit;

/**
 * CronoUnit ejemplo.
 * @author Profesor
 */
public class EjemplosChrono {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // Fecha del cumpleaños
        LocalDate cumple = LocalDate.of(1982, Month.NOVEMBER, 19);
        // Fecha de hoy
        LocalDate fechaHoy = LocalDate.now();

        // El método withYear devuelve una nueva copia de esta fecha con el
        // campo 'year' cambiado al que se pasa como parámetro.
        LocalDate proximoCumple = cumple.withYear(fechaHoy.getYear());

        // Si tu cumple ha ocurrido ya este año, añadir al año
        if (proximoCumple.isBefore(fechaHoy) || proximoCumple.isEqual(fechaHoy)) {
            proximoCumple = proximoCumple.plusYears(1);
        }

        // Con la clase Period se puede obtener la diferencia entre dos fechas
        // o utilizarlo para modificar valores de alguna fecha
        Period periodo = Period.between(fechaHoy, proximoCumple);

        // Calcular la diferencia entre la fecha de hoy y la del próximo cumple
        long per2 = ChronoUnit.DAYS.between(fechaHoy, proximoCumple);
        System.out.println("Faltan " + periodo.getMonths() + " meses, y " +
            periodo.getDays() + " días hasta tu próximo cumpleaños. (" +
            per2 + " días en total)");
    }
}
```

que mostrará por consola:

```
run:
Faltan 2 meses, y 6 días hasta tu próximo cumpleaños. (67 días en total)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Puedes consultar la API de la clase **ChronoUnit** en el siguiente enlace: [ChronoUnit API](#)

9. EXCEPCIONES

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 que falta 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 existe 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, y que no se detectan por tanto en **tiempo de compilación**, puede generar otro tipo de errores que quizá aparezcan cuando estamos usando nuestro programa, en **tiempo de ejecución**. A estos errores se les conoce como **excepciones**, y seguramente has experimentado ya los efectos de este tipo de errores en algún pequeño programa que, por ejemplo, reciba un número real o una cadena de caracteres como entrada, cuando el código está esperando un número entero. Si todavía no has experimentado esto, es un buen momento para probarlo y ver qué pasa.

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 que:

1. 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. 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, formato numérico erróneo, índice fuera de rango en un array o en una cadena, argumentos no válidos en un método, y un largo 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 sea alterado, transfiriendo el control hacia ciertas zonas o rutinas que han sido creadas previamente por el programador y cuya finalidad será el tratamiento efectivo de dichas excepciones. Estas rutinas son lanzadas (**throw**), y toman el control de la situación, de forma que permiten capturar (**catch**) la excepción producida, y darle un tratamiento adecuado, o al menos terminar lo más ordenadamente posible la ejecución del programa, en caso de que se tratara de un error irrecuperable. Si no se captura la excepción, el programa se detendrá con toda probabilidad, aunque al menos podrá dar una información precisa y detallada de qué tipo de error se ha producido, en qué línea de código, qué secuencia de instrucciones se han ejecutado hasta llegar a esa situación. Todo ello es información útil a la hora de corregir el error.

En Java, las excepciones están representadas por clases. La clase **java.lang.Exception** contiene 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**.

- La clase **Error** se encargará de los errores que se produzcan en la máquina virtual, no en nuestros programas, y que, por tanto, quedan fuera del alcance de lo que nosotros podemos corregir.
- La clase **Exception** será la que nos interese conocer, pues gestiona los errores provocados en nuestros 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**.

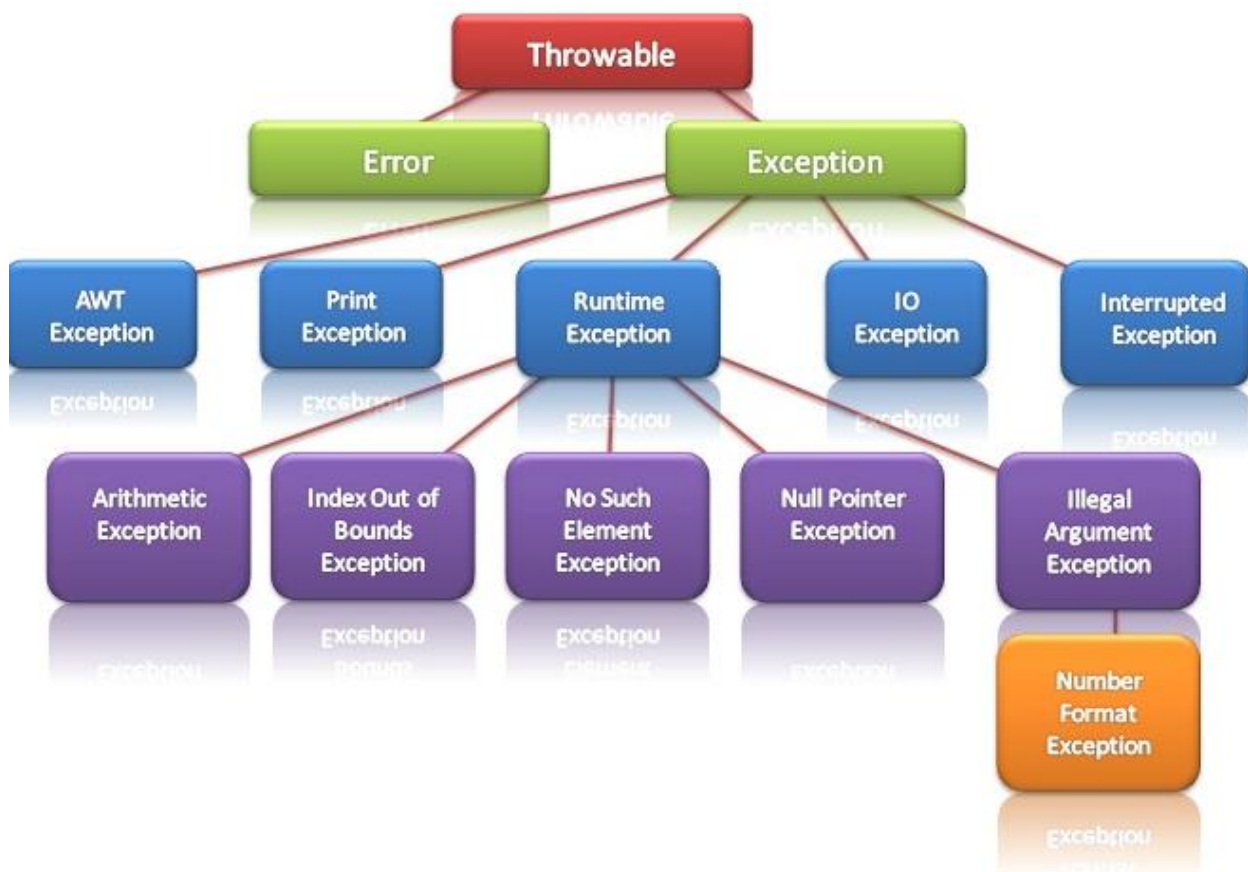
TEMA 3: UTILIZACIÓN DE OBJETOS

El programador también **puede lanzar sus propias excepciones**. Para ello se usa la cláusula **throw** (sin s, no confundir con **throws**, con s, y que se verá en el epígrafe dedicado a delegación de excepciones). El usuario puede definir excepciones como subclases de la clase **Exception**, o de alguna de alguna de sus subclases, y la cláusula **throw** fuerza la generación del error invocando al método constructor de un nuevo objeto del tipo de la excepción que se quiere lanzar... Todo ello, para ser bien entendido, requiere conocimientos de programación orientada a objetos que todavía no se han abordado en este curso, por lo que se explicará nuevamente y ya con más detalle en próximas unidades.

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**, que se dividen 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.



9.1 CAPTURAR UNA EXCEPCIÓN CON TRY.

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){
    instrucciones para manejar excepción de Tipo_excepcion_1;
}catch (Tipo_excepcion_2 objeto_excepcion){
    instrucciones para manejar excepción de Tipo_excepcion_2;
}catch (...){
    ...
}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 **ArithmeticException** 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**.

En la bibliografía sobre excepciones en Java, frecuentemente se cuestiona la utilidad de la cláusula **finally**, debido a que al ser instrucciones que se ejecutan siempre, bien podrían sacarse del bloque de manejo de la excepción. Es cierto que **finally** no aporta la posibilidad de hacer nada que no se pueda conseguir hacer de forma igualmente clara sin **finally**, por lo que en lo que a este curso respecta, su uso queda a criterio del gusto de quien programa. Eso sí, hay que saber que existe y cómo funciona.




Ejercicio Resuelto

Realiza un programa que lea por teclado un número entero empleando la clase `Scanner`. Utiliza la estructura `try-catch` para que en caso de que se introduzca un valor no esperado, como una letra y dado que saltará una excepción, se capturen las posibles excepciones y se informe del error. Si la lectura se hizo sin problemas, se mostrará el número leído.

Mostrar retroalimentación

En el siguiente fichero tienes la clase que captura las excepciones con el bloque `try-catch`.

 [Captura de la entrada desde teclado a través de excepciones](#) (4KB)

En el ejemplo que acabamos de ver se captura cualquier excepción mediante `Exception`. En esta modificación el fichero anterior, fíjate que se puede especificar según el tipo de excepción que se capture:

```
package ejemploexcepcion;

import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * Clase para entender la captura de excepciones.
 *
 * @author JJBH
 */
public class EjemploExcepcion {
    public static void main(String[] args){

        System.out.println("Escriba un número entero: ");
        try {
            Scanner teclado = new Scanner(System.in) ;
            int numero = teclado.nextInt() ;
            System.out.println("El número tecleado es: " + numero);

        } catch (InputMismatchException ex2) {
            System.err.println("Error: formato no válido.");
        }

    }
}
```

[Captura de la entrada desde teclado a través de excepciones.](#)

9.2 EL MANEJO DE EXCEPCIONES CON CATCH.

¿Es obligatorio entonces el manejo de excepciones, o podemos dejar que sea la propia máquina virtual en última instancia quien las capture y las trate?

Como hemos comentado, **siempre debemos controlar las excepciones** que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Que las excepciones las trate la propia máquina virtual, significa haber dejado que "ocurra la catástrofe", y que el programa se termine de forma abrupta, lo que nunca es deseable.

Las excepciones pueden tratarse de dos formas:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irreparable. **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 para intentar reconducir la situación.
- **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.

```

7 public static void main(String[] args){
8     boolean fueraDeLimites=true;
9     int i; //Entero que tomará valores aleatorios de 0 a 9
10    String texto[] = {"uno","dos","tres","cuatro","cinco"}; //String que representa la moneda
11
12    while(fueraDeLimites){
13        try{
14            i= (int) Math.round(Math.random()*9); //Generamos un índice aleatorio
15            System.out.println(texto[i]);
16            fueraDeLimites=false;
17        }catch(ArrayIndexOutOfBoundsException exc){
18            System.out.println("Fallo en el índice.");
19        }
20    }
21 }

```

En este ejemplo, a través de la función de generación de números aleatorios (de la clase `Math` que incorpora Java), 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, momento en que se mostrará el contenido del array para el índice elegido, y se terminará la ejecución del ciclo.

En este enlace puedes ver explicado algún ejemplo de excepciones.

[Ejemplo comentado de excepciones.](#)

Como la gestión de excepciones es un asunto bastante importante, te ponemos un enlace más con algún ejemplo adicional.

[Excepciones en Java](#)



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. Si el usuario no introduce un número en dicho rango, deberá mostrarse un mensaje de error y volver a pedir la introducción del número.

Además, utilizando el manejo de excepciones, debes controlar la entrada de dicho número y volver a solicitarlo en caso de que ésta sea incorrecta.

Por último, muestra el número leído por pantalla, junto al número de veces que el usuario ha introducido el número hasta darlo por bueno.

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.](#) (20KB)

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. Se realiza la división entera de 100 entre el número que se lea por teclado.

Como al solicitar el número pueden producirse los errores siguientes:

- ✓ Errores de entrada al introducir una letra, o bien al introducir un número real, etc. Por ello capturaremos la excepción `InputMismatchException` generada.
- ✓ Errores aritméticos al intentar dividir por 0, si el usuario introduce un 0 como entrada. Por ello capturaremos a través de la excepción `ArithmeticException` generada.

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.

[Filtrado de entrada desde teclado a través de excepciones.](#)