Fundamentos de la programación en JAVA

Departamento de Formación

Ingeniería de Software Avanzado

03/03/2010

Contenido

1. ¿Qué es Java?

Java fue diseñado en 1990 por James Gosling, de Sun Microsystems, como software para dispositivos electrónicos de consumo. Curiosamente, todo este lenguaje fue diseñado antes de que diese comienzo la era World Wide Web, puesto que fue diseñado para dispositivos electrónicos como calculadoras, microondas y la televisión interactiva.

En los primeros años de la década de los noventa, Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Tras unos comienzos dudosos, Sun decidió crear una filial, denominada FirstPerson Inc., para dar margen de maniobra al equipo responsable del proyecto.

Inicialmente Java se llamó Oak (roble en inglés), aunque tuvo que cambiar el nombre, debido a que ya estaba registrado por otra Empresa. Se dice que este nombre se le puso debido a la existencia de tal árbol en los alrededores del lugar de trabajo de los promotores del lenguaje.

Tres de las principales razones que llevaron a crear Java son:

- Creciente necesidad de interfaces mucho más cómodas e intuitivas que los sistemas de ventanas que proliferaban hasta el momento.
- Fiabilidad del código y facilidad de desarrollo. Gosling observó que muchas de las características que ofrecían C o C++ aumentaban de forma alarmante el gran coste de pruebas y depuración. Por ello en sus ratos libres creó un lenguaje de programación donde intentaba solucionar los fallos que encontraba en C++.
- Enorme diversidad de controladores electrónicos. Los dispositivos electrónicos se controlan mediante la utilización de microprocesadores de bajo precio y reducidas prestaciones, que varían cada poco tiempo y que utilizan diversos conjuntos de instrucciones. Java permite escribir un código común para todos los dispositivos.

Por todo ello, en lugar de tratar únicamente de optimizar las técnicas de desarrollo y dar por sentada la utilización de C o C++, el equipo de Gosling se planteó que tal vez los lenguajes existentes eran demasiado complicados como para conseguir reducir de forma apreciable la complejidad de desarrollo asociada a ese campo. Por este motivo, su primera propuesta fue idear un nuevo lenguaje de programación lo más **sencillo** posible, con el objeto de que se pudiese adaptar con facilidad a cualquier entorno de ejecución.

El proyecto Green fue el primero en el que se aplicó Java, y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar. Con este fin se construyó un ordenador experimental denominado **7 (Star Seven)*. El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. En el sistema aparecía ya *Duke*, la actual mascota de Java.

Más tarde Java se aplicó a otro proyecto denominado *VOD (Video On Demand)* en el que se empleaba como interfaz para la televisión interactiva que se pensaba iba a ser el principal campo de aplicación de Java. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo.

Una vez que en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, instaron a FirstPerson a desarrollar nuevas estrategias que produjeran beneficios. Entre ellas se encontraba la aplicación de Java a Internet, la cual no se consideró productiva en ese momento.

Aunque muchas de las fuentes consultadas señalan que Java no llegó a caer en un olvido, lo cierto es que tuvo que ser Bill Joy (cofundador de Sun y uno de los desarrolladores principales del sistema operativo Unix de Berckley) el que sacó a Java del letargo en que estaba sumido. Joy juzgó que Internet podría llegar a ser el campo adecuado para disputar a Microsoft su supremacía en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes.

Para poder presentarlo en sociedad se tuvo que modificar el nombre de este lenguaje de programación y se tuvo que realizar una serie de modificaciones de diseño para poderlo adaptar al propósito mencionado. Así *Java fue presentado en sociedad en agosto de 1995.*

Algunas de las razones que llevaron a Bill Joy a pensar que Java podría llegar a ser rentable son:

- Java es un *lenguaje orientado a objetos*: Esto es lo que facilita abordar la resolución de cualquier tipo de problema.
- Es un lenguaje *sencillo*, aunque sin duda potente.
- La ejecución del código Java es *segura y fiable*: Los programas no acceden directamente a la memoria del ordenador, siendo imposible que un programa escrito en Java pueda acceder a los recursos del ordenador sin que esta operación le sea permitida de forma explícita. De este modo, los datos del usuario quedan a salvo de la existencia de virus escritos en Java. La ejecución segura y controlada del código Java es una característica única, que no puede encontrarse en ninguna otra tecnología.
- Es totalmente *multiplataforma*: Es un lenguaje sencillo, por lo que el entorno necesario para su ejecución es de pequeño tamaño y puede adaptarse incluso al interior de un navegador.

Existen muchas críticas a Java debido a su *lenta velocidad* de ejecución, aproximadamente unos 20 veces más lentos que un programa en lenguaje C. Sun está trabajando intensamente en crear versiones de Java con una velocidad mayor.

El problema fundamental de Java es que utiliza una representación intermedia denominada *código de byte* para solventar los problemas de portabilidad. Los *códigos de byte* posteriormente se tendrán que transformar en código máquina en cada máquina en que son utilizados, lo que ralentiza considerablemente el proceso de ejecución.

La solución que se deriva de esto parece bastante obvia: fabricar ordenadores capaces de comprender directamente los códigos de byte. Éstas serían unas máquinas que utilizaran Java como sistema operativo y que no requerirían en principio de disco duro porque obtendrían sus recursos de la red.

A los ordenadores que utilizan Java como sistema operativo se les llama Network Computer, WebPC o WebTop. La primera gran empresa que ha apostado por este tipo de máquinas ha sido Oracle, que en enero de 1996 presentó en Japón su primer NC (Network Computer), basado en un procesador RISC con 8 Megabytes de RAM. Tras Oracle, han sido compañías del tamaño de Sun, Apple e IBM las que han anunciado desarrollos similares.

1.1. Características de Java

Los creadores de Java diseñaron el lenguaje con las siguientes ideas en mente:

- **Simplicidad**

Java está basado en C++, por lo que, si se ha programado en C o en C++, el aprendizaje de la sintaxis de Java es casi inmediato. Sin embargo, se modificaron o eliminaron ciertas características que en C++ son fuente de problemas, como la aritmética de punteros, las estructuras, etc. Además, no debemos preocuparnos de la gestión de la memoria. Java se ocupa de descargar los objetos que no utilicemos. Es prácticamente imposible, por ejemplo, escribir en una posición de memoria de otro programa.

- **Orientación de Objetos (OO)**

Java es un lenguaje de programación completamente Orientado a Objetos, que nos permite llevar a cabo el desarrollo de aplicaciones de manera rápida y sencilla para manejar un ambiente de desarrollo mucho más sencillo. Java permite pasar de una etapa de análisis y diseño, a un desarrollo muy rápido y posiblemente con muy pocas modificaciones.

- **Distribuido**

Java posee una extensa colección de herramientas que proporcionan la capacidad de trabajar en red de forma simple y robusta.

- **Robusto**

Java permite escribir programas fiables con mucho menor esfuerzo que en otros lenguajes. El compilador detecta problemas, como la sobrescritura de posiciones de memoria, que en otros lenguajes aparecerían en tiempo de ejecución. Además, la eliminación del uso de punteros en elementos como cadenas de caracteres o arrays evita muchos problemas que son comunes (y difíciles de depurar) en C o C++.

- **Multithread**

Java, al igual que C o C++, permite trabajar con varios hilos de ejecución simultáneos, facilitando la programación en sistemas multiprocesador, y mejorando el funcionamiento en tiempo real.

- **Seguro**

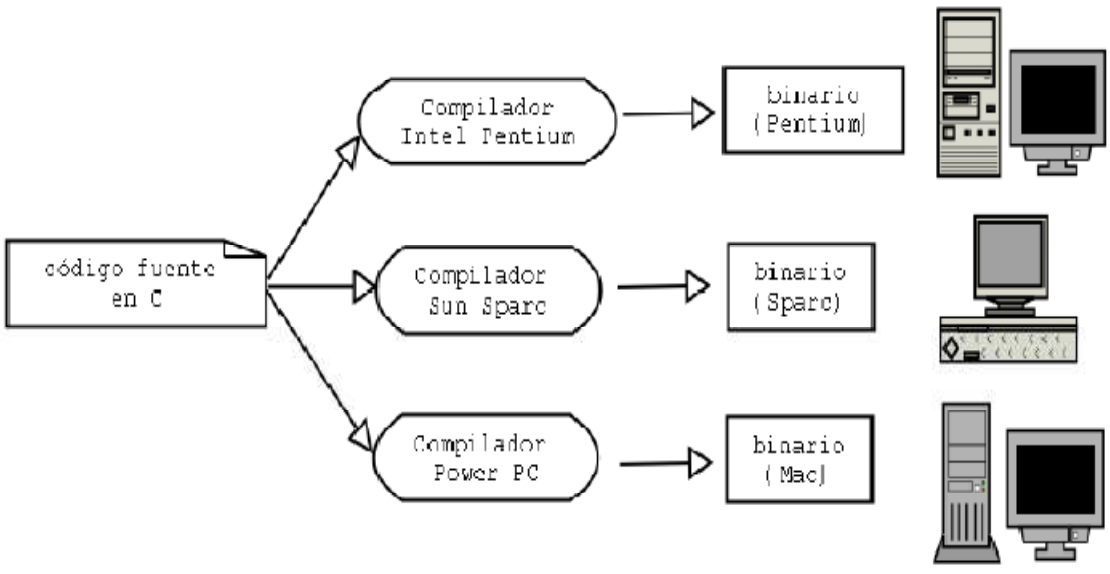
Java está pensado para ser utilizado en red, por lo que se ha cuidado mucho la seguridad. En principio, se supone que es capaz de evitar que se rebase la pila del sistema en tiempo de ejecución, que se corrompa la memoria externa a un proceso, o que se pueda acceder a ficheros locales de un ordenador que está ejecutando un applet en su navegador, por ejemplo.

- **Independencia de plataforma**

He aquí una de las características más importantes y conocidas de Java. Un programa en Java sigue la **filosofía WORE** (*Write Once*, *Run Everywhere*), es decir, que una vez escrito, puede ejecutarse en cualquier plataforma hardware con cualquier sistema operativo sin recompilar el código.

¿Qué quiere decir esto?. En la compilación tradicional de programas escribimos nuestro código fuente pensando en el sistema operativo en el que va a ejecutarse, ya que cada S.O. tiene sus propias peculiaridades, librerías a las que es necesario invocar, etc. No puede escribirse con el mismo código un programa para Linux y para Windows, aunque corran en la misma máquina. Existe, por tanto, **dependencia a nivel de código fuente**.

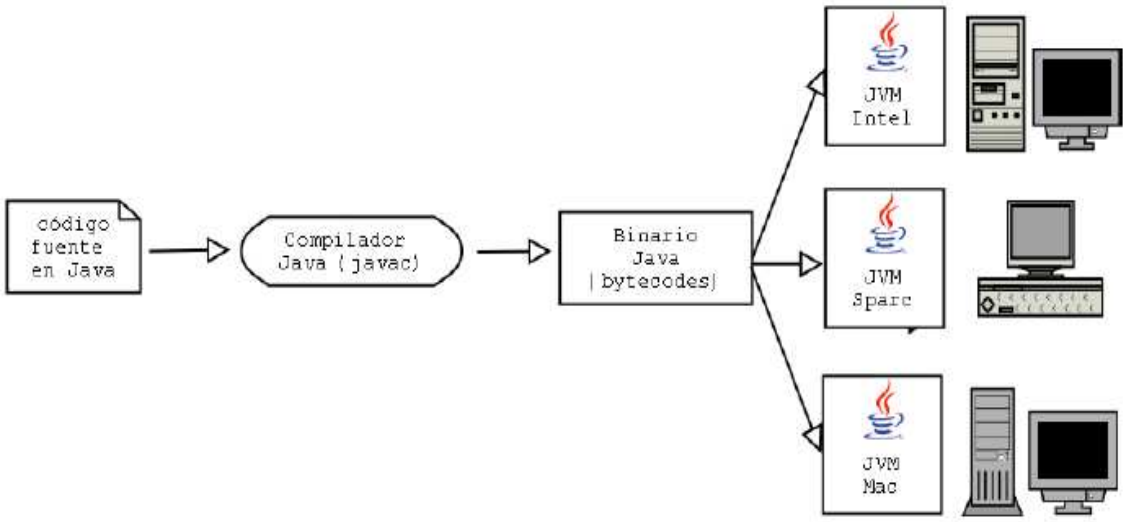
Una vez tenemos escrito nuestro programa, lo compilamos (*Imagen 1*). El compilador traduce el código fuente a código máquina capaz de ser entendido por el procesador de la máquina en la que va a correr ese programa. Es decir, que tenemos **dependencia a nivel de archivo binario**, ya que cada compilador es específico de cada arquitectura. Un programa compilado para una máquina Intel no funcionará en un PowerPC, ni en una Sparc.



1.1.1. El software Development Kit de Java

Java elimina estas dependencias. Una vez que escribamos y compilemos nuestro código fuente, podremos llevar el archivo binario a cualquier ordenador que tenga instalado una **Máquina Virtual de Java** (*JVM, Java Virtual Machine*) y se ejecutará exactamente igual, independientemente de la arquitectura software y hardware de ese ordenador.

¿Y qué es la JVM?. Fijándonos en la *Imagen 2*, en la que se muestra la compilación de un programa Java. Comenzamos escribiendo nuestro código fuente Java. No nos tenemos que preocupar de las peculiaridades del S.O. ni del ordenador en el que se vaya a ejecutar ese código. Una vez escrito, lo compilamos con el compilador de Java, que nos genera un archivo de *bytecodes*. Los *bytecode* son una especie de *código intermedio*, un conjunto de instrucciones en un lenguaje máquina independiente de la plataforma.



Cuando queramos ejecutar nuestro programa, la JVM instalada en el ordenador leerá el archivo de *bytecodes* y lo interpretará **en tiempo de ejecución**, traduciéndolo al código máquina nativo de la máquina en la que se está ejecutando en ese momento. Por tanto, la JVM es un intérprete de *bytecodes*.

En ocasiones, este sistema puede resultar lento, aunque, paulatinamente se está aumentando la velocidad del intérprete, la ejecución de programas Java será más lenta que la de programas compilados en código nativo de la plataforma. Existen, sin embargo, compiladores JIT (*Just In Time*), que lo que hacen es interpretar los *bytecodes* la primera vez que se ejecuta el programa, guardando el código nativo resultante, y usando este en las demás invocaciones del programa. De este modo, se puede llegar a incrementar entre 10 y 20 veces la velocidad respecto al intérprete estándar de Java.

1.2. El software Development Kit de Java

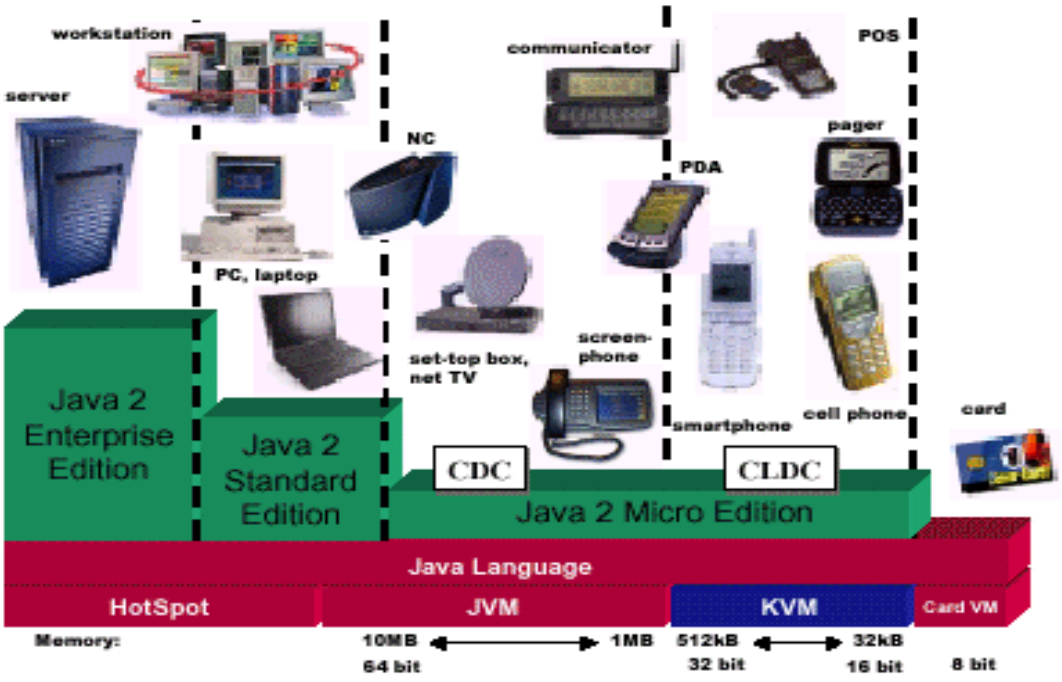
El JDK (*Java Development Kit*), también llamado SDK (*Software Development Kit*, Kit de Desarrollo de Software) de Java está compuesto por el conjunto de herramientas necesarias para compilar y ejecutar código escrito en Java. Comprende, principalmente, el compilador (*javac*), la JVM, y el conjunto de paquetes de clases que forman una base sobre la que programar.

Existen tres ediciones del SDK:

- **J2SE (*Java 2 Standard Edition*):** Versión estándar de Java, con la que trabajaremos. Lo de Java 2 es cosa del departamento de marketing de Sun: a partir de la versión 1.2 del SDK, Java pasó a llamarse Java 2, para denotar una importante evolución en la plataforma.

- **J2ME (*Java 2 Mobile Edition*):** Versión de Java orientada a dispositivos móviles y pequeños, como PDAs o teléfonos móviles.

- **J2EE (*Java 2 Enterprise Edition*):** Versión orientada al entorno empresarial. Se utiliza, principalmente, en aplicaciones de servidor, como *servlets*, EJBs (*Enterprise Java Beans*) y JSPs (*Java Server Pages*).



1.3. Versiones de Java

- **Java 1**

  - *Java 1.0* (Enero 1996) - 8 paquetes, 212 clases - Primera versión pública. La presión hizo que se hiciera pública demasiado pronto, lo cual significa que el diseño del lenguaje no es demasiado bueno y hay muchos errores. Respecto a seguridad, es restrictivo por defecto.

  - *Java 1.1* (Marzo 1997) - 23 paquetes, 504 clases - mejoras de rendimiento en la JVM, nuevo modelo de eventos en AWT, clases anidadas, serialización de objetos, API de JavaBeans, archivos jar, internacionalización, API Reflection (Reflexión), JDBC (Java Data base Connectivity), RMI (Remote Method Invocation). Se añade la firma del código y la autentificación. Es la primera versión lo suficientemente estable y robusta.

- **Java 2**

  - *Java 1.2* (Diciembre 1998) - 59 paquetes, 1520 clases - JFC (Swing), Drag and Drop, Java2D, Corba, API Collections. Se producen notables mejoras a todos los niveles. Para enfatizar esto Sun lo renombra como Java 2. El JDK (Java Development Kit) se renombra como SDK (Software Development Kit). Se divide en J2SE, J2EE y J2ME.

  - *Java 1.3* (Abril 2000) - 77 paquetes, 1595 clases - Orientada sobre todo a la resolución de errores y a la mejora del rendimiento; se producen algunos cambios menores como la inclusión de JNDI (Java Naming and Directory Interface) y la API Java Sound. También incluye un nuevo compilador de alto rendimiento JIT (Just In Time).

- *Java 1.4* (2002) - 103 paquetes, 2175 clases - También conocido como Merlin, es la versión actual. Mejora notablemente el rendimiento y añade entre otros soporte de expresiones regulares, una nueva API de entrada/salida de bajo nivel (NIO, New I/O), clases para el trabajo con Collections, procesado de XML; y mejoras de seguridad como el soporte para la criptografía mediante las Java Cryptography Extension (JCE), la inclusión de la Java Secure Socket Extension (JSSE) y el Java Authentication and Authorization Service (JAAS).

- *Java 1.5* (Octubre 2004) - 131 paquetes, 2656 clases - También conocido como Tiger, renombrado por motivos de marketing como Java 5.0. Incluye como principales novedades:
  - Tipos genéricos (generics)
  - Autoboxing/unboxing conversiones implícitas entre tipos primitivos y los wrappers correspondientes.
  - Enumeraciones.
  - Bucles simplificados
  - printf.
  - Funciones con número de parámetros variable (Varargs)
  - Metadatos en clases y métodos.

1.4. Instalación y configuración del Kit J2SE

Para la parte de la instalación necesitamos descargarnos el ejecutable de j2se, podemos acceder a la página de Sun (http://java.sun.com/) y descargarlo.

Encontraremos versiones para varios sistemas operativos y plataformas. Nótese que, en algunos casos, existen dos modalidades del J2SE para la misma plataforma: con o sin NetBeans. NetBeans es un IDE, o Entorno de Desarrollo Integrado, que nos permite programar aplicaciones más cómodamente, incorporando un editor con resaltado de texto, herramientas para facilitar la programación gráfica y de depurado, etc. Sin embargo, no es necesario descargarlo para trabajar con el SDK. Por ello se nos da la opción de descargar el J2SE sin el IDE.

La instalación consistirá en algo tan sencillo como hacer doble click sobre un icono (en Windows) o ejecutar un script desde la línea de comandos (en UNIX/Linux). Una vez terminada la instalación, sería conveniente poder ejecutar tanto el compilador como la JVM desde cualquier directorio, no sólo el de instalación del J2SE. Si, abriendo una ventana de línea de comandos (tanto en Windows como en UNIX) escribimos java, y nos da un error, indicando que no se reconoce el comando, será necesario configurar correctamente el PATH del sistema. Por ejemplo, para los siguientes casos:

- **Windows 98/ME**: En el Autoexec.bat, añadir la línea SET PATH=c:\j2sdk1.X.Y\bin;%PATH% (suponiendo que ese es el directorio en el que está instalado el SDK, claro).

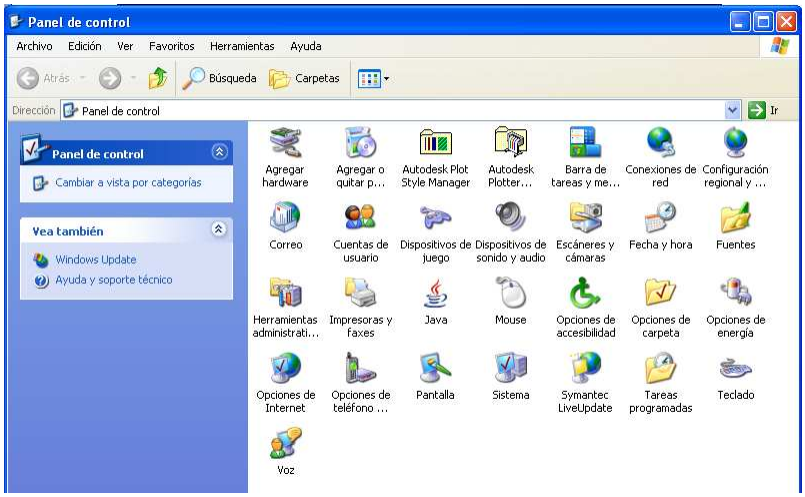
- **Windows NT/2000/XP**: En Panel de Control/Sistema/Avanzado, pulsar sobre el botón Variables de Entorno y, en la lista Variables del Sistema localizar la entrada PATH y añadir la ruta del SDK.

- **Linux**: Añadir en el archivo $HOME/.bashrc o $HOME/.bash_ profile la línea export PATH=/usr/ local/j2sdk1.X.Y/bin:$PATH

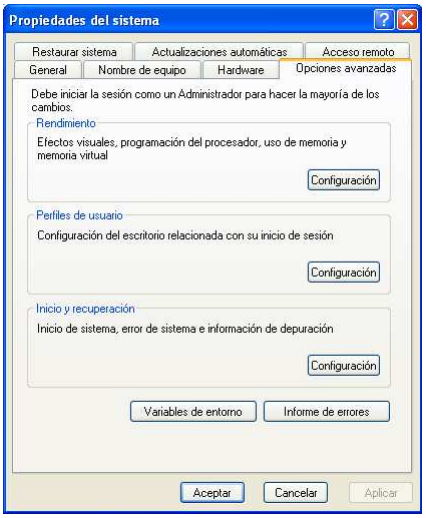
NOTA: X, Y son la versión y la subversión del j2se descargado.

Gráficamente para el apartado **Windows NT/2000/XP, sería**:

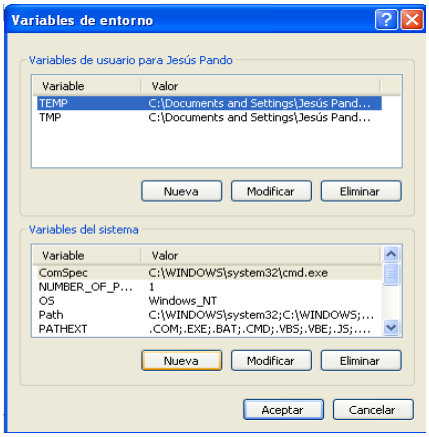
a. Accedemos al panel de control:

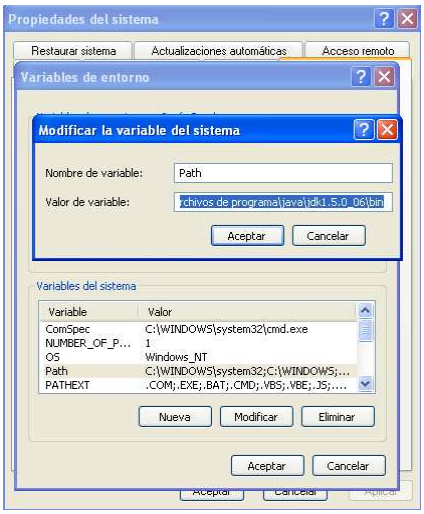


b. Se selecciona ***Sistema > Opciones Avanzadas:***

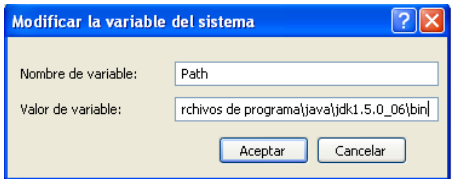


c. Se selecciona en **Variables** del sistema la variable **Path** y después se elige el botón **Modificar** (si no existe, se crea una nueva con ese mismo nombre):





d. Se escribe la ruta correcta de dónde instalamos el software de java.



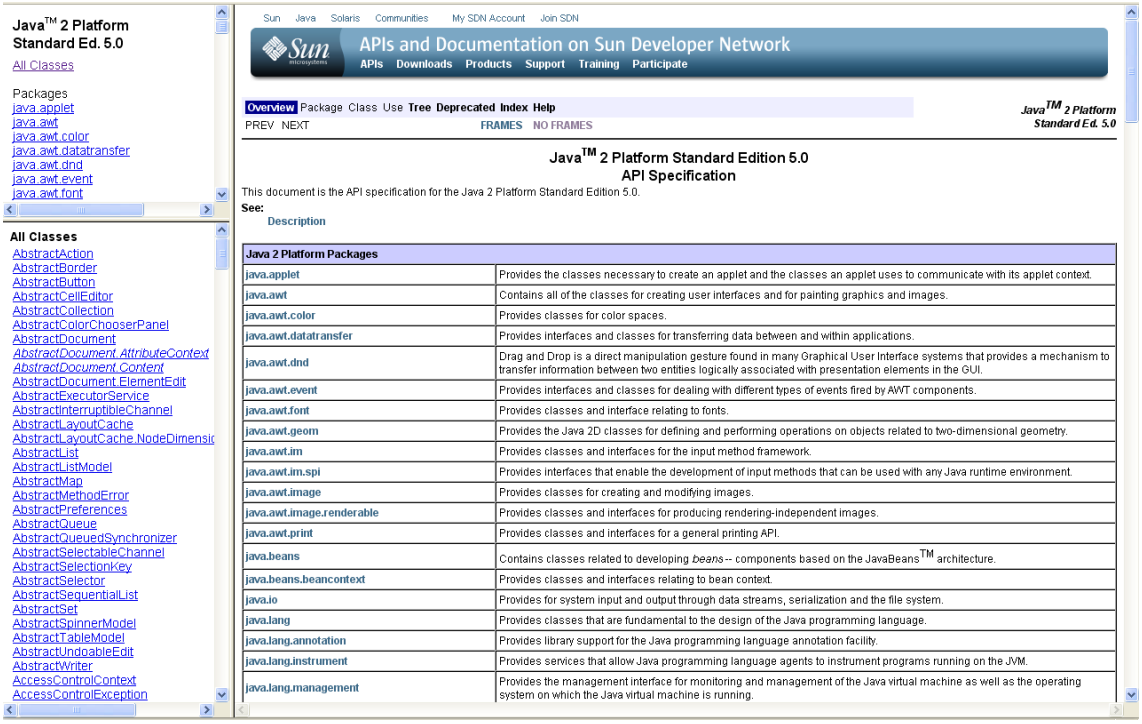
Por ejemplo:

c:\archivos de programa\java\jdk1.5.0_06\bin (iniciando con un punto y coma para separar la configuración anterior).

1.5. El API del SDK

En cuanto tengamos la JVM funcionando, y queramos comenzar a programar código, nuestra primera pregunta será: ¿y cómo sé qué clases y funciones proporciona Java?. Es típico que queramos realizar tareas como, por ejemplo, ordenar un array de datos, y no sabemos si Java implementa esa función o tenemos que programarla nosotros mismos. Para ello, Sun proporciona la documentación del API (Interfaz de Programación de Aplicaciones) para consultarla online en http://java.sun.com/j2se/1.X.Y/docs/api/.

Si queremos descargarla para disponer de ella en nuestro ordenador, podemos bajarla de la dirección donde se encuentra el SDK. Como se ve en la figura, el API está en formato html, y muestra todas las clases, métodos y atributos que proporciona Java.



NOTA: Una API (*Application Programming Interface*) constituye un conjunto de rutinas, procedimientos, protocolos, funciones y herramientas que una determinada biblioteca pone a disposición para que sean utilizados por otro software como una capa de abstracción.

1.6. Como compilar archivos

Los archivos con código fuente en Java tienen siempre la extensión ***.java***. Compilarlos, suponiendo que nos encontramos en el mismo directorio que el fichero fuente, es tan sencillo como:

```
javac archivo.java
```

que nos creará uno o varios archivos con extensión ***class***. Esos archivos serán nuestro programa compilado, que podrá entender y ejecutar la JVM. El compilador posee muchas opciones. Bajo Unix/Linux pueden consultarse con *man javac*. Algunas de las más utilizadas son:

```
javac -d directorio archivo.java
```

que nos permite compilar el fichero fuente y depositar la clase compilada en el directorio especificado. Esto es útil si la clase pertenece a un paquete y queremos, por tanto, depositar la clase en una estructura de directorios y subdirectorios acorde a la del paquete. Si el directorio no existe, no lo creará, sino que nos dará un error. El modo de que nos cree previamente el(los) directorio(s) es con la sentencia:

```
javac -d . archivo.java
```

que leerá de archivo.java la estructura de directorios que componen el paquete, creará esos directorios y depositará la clase compilada allí.

Otra opción interesante es:

```
javac -classpath classpath archivo.java
```

que permite redefinir el CLASSPATH, ignorando el definido por defecto para la máquina o usuario.

1.7. El *Classpath*

Cuando compilamos una clase Java, ésta necesitará importar otras clases, ya sean del propio j2sdk, o escritas por terceros. Para que pueda encontrarlas, es necesario definir una variable en el sistema que contenga las rutas en las que el compilador debe buscar las clases (de igual manera que hicimos con la variable PATH).

Al instalar el j2sdk se definirá un classpath por defecto. Sin embargo, podría suceder que necesitáramos redefinir esas rutas para buscar clases en otra parte de nuestra máquina. Usaremos la opción *-classpath*, como se especificó en el apartado anterior. Por ejemplo:

```
javac -classpath /ejemplos:.:/lib/milibreria.jar archivo.java
```

NOTA:

a) Podemos observar que las diferentes rutas se separan mediante dos puntos (":"). Para especificar el directorio en el que nos encontremos al invocar a *javac*, se utiliza el punto (".").

b) Para no complicarnos a la hora de compilar y ejecutar nuestras clases, la solución más fácil es definir la variable CLASSPATH dentro de nuestras variables de entorno otorgándole el valor . (punto), de esta manera buscará todos los valores que necesite desde el directorio raíz.

1.8. Ejecución de programas en Java

Si tenemos una clase llamada MiClase.class, la ejecutaremos escribiendo:

```
java MiClase
```

Fallos típicos al ejecutar un programa:

- Estamos intentando ejecutar una clase que no tiene definido un método *main().*

- Hemos escrito *java MiClase.class*. El tipo (class) no se incluye.

- Al intentar ejecutar una clase llamada MiClase.class, java arroja un error de tipo *java.lang.NoClassDefFoundError*, a pesar de que estamos seguros de haber especificado correctamente el directorio en el que está la clase. Probablemente se deba a que esa clase pertenece a un paquete. Comprobemos el código fuente. Si incluye al principio una línea del estilo a:

  ```
  package otrasclases.misclases;
  ```

  entonces la clase debe encontrarse en una estructura de directorios con esos nombres. Podemos, por ejemplo, crear un subdirectorio del directorio actual al que llamaremos *otrasclases*.

  A continuación, dentro de ese, crearemos otro de nombre *misclases*. Y dentro de él copiaremos la clase que estábamos tratando de ejecutar. Si ahora ejecutamos, desde nuestro directorio actual, *java otrasclases.misclases.MiClase*, debería funcionar.

1.9. Empaquetar clases (JAR)

Cuando tenemos un programa grande, con varios paquetes y clases, ya sabemos cómo organizarlo, compilarlo y ejecutarlo. Sin embargo, si queremos instalarlo en otro ordenador puede resultarnos un poco tedioso. Tenemos que llevarnos directorios enteros, con los ficheros que hay dentro y demás.

Lo ideal es meter todos estos ficheros y directorios en un único fichero comprimido. Java, con su comando *jar*, que está en el directorio *bin* de donde tengamos java, nos permite hacer esto. Empaqueta todo lo que le digamos (directorios, clases, ficheros de imagen o lo que queramos) en un único fichero de extensión *.jar*. Un fichero de extensión .jar es similar a los .zip de Winzip, a los .rar de Winrar o a los ficheros .tar del tar de unix.

Java nos da además otra opción, podemos ejecutar las clases del fichero .jar sin tener que desempaquetarlo. Simplemente podemos enviar a quien nosotros queramos el fichero .jar con todo dentro y ya está listo para ejecutar.

1.9.1 Crear un fichero JAR

Para crear un fichero jar, en primer lugar tenemos que tener todo ya perfectamente preparado y compilado, funcionando. Tenemos diferentes posibilidades:

- Si las clases de nuestro programa no pertenecen a paquetes, simplemente debemos meter las clases en el fichero .jar Para ello, vamos al directorio donde estén los ficheros .class y ejecutamos el siguiente comando:

```
cd directorio_con_los_class

jar -cf fichero.jar fichero1.class fichero2.class
```

  Donde:

  - La opción "**c**" indica que queremos crear un fichero.jar nuevo. Si ya existía, se machacará, así que hay que tener cuidado.
  - La opción "**f**" sirve para indicar el nombre del fichero, que va inmediatamente detrás. En nuestro caso, **fichero.jar**.
  - Finalmente se pone una lista de ficheros .class (o de cualquier otro tipo) que queramos meter en nuestro jar. Se pueden usar comodines, estilo *.class para meter todos los .class de ese directorio.

- Si las clases de nuestro programa pertenecen a paquetes, debemos meter en nuestro jar la estructura de directorios equivalente a los paquetes entera. Para ello, nos vamos al *directorio padre* de donde empiece nuestra estructura de paquetes. Imaginemos el caso que queremos hacer un HolaMundo.java, el cual vamos a empaquetar dentro de un paquete llamado prueba, el comando a ejecutar sería:

```
cd directorio_padre_de_prueba

jar -cf fichero.jar prueba
```

Las opciones son las mismas, pero al final en vez de las clases, hemos puesto el nombre del directorio. Esto meterá dentro del jar el directorio y todo lo que hay debajo.

Otra opción sería meter los .class, pero indicando el camino relativo para llegar e ellos:

```
cd directorio_padre_de_prueba

jar -cf fichero.jar prueba\HolaMundo.class
```

NOTA: Cuidado porque la barra en Windows va a revés

1.9.2 Listar un fichero JAR

Para comprobar si nuestro jar está bien hecho, podemos ver su contenido. El comando sería:

```
jar -tf fichero.jar
```

- La opción "**t**" indica que queremos un listado del fichero.jar.
- La opción "**f**" es igual que antes. Esto nos dará un listado de los class (y demás ficheros) que hay dentro, indicando en que directorio están. Deberíamos comprobar en ese listado que están todas las clases que necesitamos y la estructura de directorios concuerda con la de paquetes.

1.9.3 Modificar un fichero JAR

Para cambiar un fichero dentro de un jar o añadirle uno nuevo, la opción del comando jar es "**u**". Si el fichero existe dentro del jar, lo reemplaza. Si no existe, lo añade.

Por ejemplo, si hacemos un cambio en nuestro HolaMundo.class con paquete y lo recompilamos, podemos reemplazarlo así en el jar

```
jar -uf fichero.jar prueba\HolaMundo.class
```

1.9.4 Como ejecutar un JAR

Para ejecutar un jar, simplemente debemos poner el fichero jar en el *CLASSPATH*. Ojo, hay que poner el fichero.jar, NO el directorio en el que está el fichero.jar. Este suele ser un error habitual al empezar, pensar que basta con poner el directorio donde está el jar. Para los .class, basta poner el directorio, para los .jar hay que poner el fichero.jar

El path para indicar la ubicación del fichero puede ser absoluto o relativo al directorio en el que ejecutemos el comando java. El comando para ejecutar una clase dentro de un jar, en nuestro caso del HolaMundo con paquete, suponiendo que estamos en el directorio en el que está el fichero.jar, sería este

```
java -cp .\fichero.jar prueba.HolaMundo
```

Simplemente, en la opción -cp del CLASSPATH hemos puesto el fichero.jar con su PATH relativo. Detrás hemos puesto el nombre de la clase, completo, con su paquete delante.

1.9.5 El fichero MANIFIESTO

Ejecutar así tiene una pega. Además de acordarse de poner la opción *-cp*, hay que saber el nombre de la clase que contiene el método *main()*. Además, si nuestro programa es muy grande, tendremos varios jar, tanto nuestros como otros que nos bajemos de internet o de donde sea. La opción -cp también puede ser pesadita de poner en ocasiones.

Una opción rápida que a todos se nos ocurre es crearse un pequeño fichero de script/comandos en el que se ponga esta orden. Puede ser un fichero .bat de windows o un script de unix. Este fichero debe acompañar al fichero.jar y suponiendo que estén en el mismo directorio, su contenido puede ser este

```
java -cp .\fichero.jar prueba.HolaMundo
```

Para ejecutarlo, se ejecuta como un fichero normal de comandos/script. Si el fichero se llama ejecuta.sh o ejecuta.bat, según sea unix o windows:

```
$ ./ejecuta.sh
Hola Mundo

C:\> ejecuta.bat
Hola Mundo
```

Sin embargo, java nos ofrece otra posibilidad de forma que no tengamos que hacer este fichero. Simplemente, en un fichero de texto metemos una línea en la que se ponga cual es la clase principal. Este fichero se conoce como *fichero de manifiesto* y su contenido puede ser este

```
Main-Class: prueba.HolaMundo
```

(muy importante el espacio en blanco después de los dos puntos)

Cuando construimos el jar, debemos incluir este fichero de una forma especial. Por ejemplo, si el fichero lo llamamos manifiesto.txt y lo ponemos en el directorio donde vamos a construir el jar, el comando para hacerlo sería este

```
jar cmf manifiesto.txt fichero.jar prueba\HolaMundo.class
```

Al comando de crear jar le hemos añadido la opción "**m**" para indicar que vamos a añadir un fichero de manifiesto. Hemos añadido además el fichero manifiesto.txt. El orden de las opciones "**mf**" es importante. El fichero de manifiesto y el fichero.jar se esperan en el mismo orden que pongamos las opciones. En el ejemplo, como hemos puesto primero la opción "**m**", debemos poner manifiesto.txt delante de fichero.jar. El resto de ficheros son los que queremos empaquetar.

Una vez construido, se ejecuta fácilmente. Basta con poner

```
java -jar fichero.jar
```

La opción "**-jar**" indica que se va a ejecutar el fichero.jar que se ponga a continuación haciendo caso de su fichero de manifiesto. Como este fichero de manifiesto dice que la clase principal es **prueba.HolaMundo**, será esta la que se ejecute.

De esta forma nos basta con entregar el jar y listo. El comando para arrancarlo es sencillo.

Es más, en windows, si lo configuramos para que los ficheros jar se abran con java y la opción -jar, bastará con hacer doble click sobre ellos para que se ejecuten.

2. Programación Orientada a Objetos

La programación orientada a objetos nació con posterioridad a la programación procedimental. Esta última toma como unidades organizacionales a los procedimientos (de ahí su nombre), mientras que en la primera las unidades organizacionales son los objetos. Pero, ¿qué es un objeto?.

En la vida real podemos observar que estamos rodeados de objetos, por ejemplo una mesa, una silla, un bolígrafo, etc. y todos ellos podrían clasificarse (con mayor o menor detalle) atendiendo a una definición. Además, sabemos que a pesar de existir muchos tipos de mesas, todas ellas tienen unas características comunes.

Esta idea fue trasladada a la informática y surgieron los conceptos de **clase** y **objeto**. Podemos decir que una clase es un concepto sobre una entidad abstracta que define cómo serán todos los objetos que existan de ese tipo. Por tanto, un objeto es una concreción mientras que una clase es una abstracción.

Si trasladamos la definición anterior a la jerga técnica informática, podemos decir que una clase es un prototipo que define las propiedades y los métodos comunes a múltiples objetos de un mismo tipo. Sería como una plantilla para la creación de objetos. Por su parte, un objeto es un conjunto de propiedades y métodos capaces de manipular dichas propiedades. No vamos a entrar en los detalles concretos en Java ya que veremos todo esto en temas posteriores.

Para comprender bien la POO debemos olvidar un poco la Programación Estructurada, que si nos fijamos bien es algo artificial, la POO es una forma de abordar los problemas más naturales. Aquí natural significa más en contacto con el mundo real que nos rodea, de esta forma si queremos resolver un problema determinado, debemos identificar cada una de las partes del problema con objetos presentes en el mundo real.

En esta definición de POO ya estamos haciendo referencia al elemento clave de la misma: el objeto. El objeto va a ser la modelización de los objetos que nos encontramos en el mundo real, estos objetos los vamos a utilizar en nuestros programas para dar la solución al problema que nos ocupe en cada caso.

2.1 Objeto

Como ya hemos adelantado un objeto es la pieza básica de la POO, es una representación o modelización de un objeto real perteneciente a nuestro mundo, por ejemplo, podemos tener un objeto perro que represente a un perro dentro de nuestra realidad, o bien un objeto factura, cliente o pedido.

Los objetos en la vida real tienen todos, dos características: **estado y comportamiento**.

- El estado de un objeto viene definido por una serie de parámetros que lo definen y que lo diferencian de objetos del mismo tipo. En el caso de tener un objeto perro, su estado estaría definido por su *raza, color de pelo, tamaño*, etc.
- Y el comportamiento viene definido por las *acciones* que pueden realizar los objetos, por ejemplo, en el caso del perro su comportamiento sería: saltar, correr, ladrar, etc. El comportamiento permite distinguir a objetos de distinto tipo, así por ejemplo el objeto perro tendrá un comportamiento distinto a un objeto gato.

Los parámetros o variables que definen el estado de un objeto se denominan *atributos* o *variables miembro* y las acciones que pueden realizar los objetos se denominan *métodos* o funciones miembro, y para indicar variables miembro y funciones miembro se utiliza el término general miembro.

Si lo comparamos con la programación estructurada podríamos hacer la siguiente aproximación: los atributos o variables miembro serían variables y los métodos o funciones miembro procedimientos y funciones.

Los *atributos* de un objeto deben encontrarse *ocultos* al resto de los objetos, es decir, no se va a poder acceder directamente a los atributos de un objeto para modificar su estado o consultarlo. Para *acceder* a los atributos de un objeto se deben utilizar *métodos*. Es decir, los métodos exponen toda la funcionalidad del objeto, mientras que los detalles del estado interno del objeto permanecen ocultos. Incluso algunos métodos también pueden permanecer ocultos.

El hecho de ocultar la implementación interna de un objeto, es decir, como está construido y de que se compone se denomina **encapsulación**. La encapsulación es uno de los beneficios y particularidades del paradigma de la Programación Orientada a Objetos.

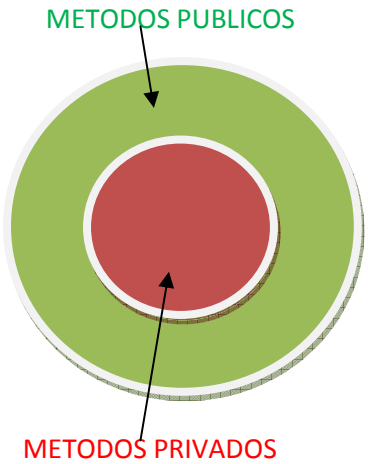
Normalmente un objeto ofrece una *parte pública* que será utilizada por otros objetos para interactuar entre sí, pero también permanece una *parte oculta* para encapsular los detalles de la implementación del objeto.

Ya se ha dicho que un objeto está compuesto de atributos y métodos. Como la caja negra de un avión, el objeto recubre la información que almacena y solamente podemos obtener la información e indicarle que realiza acciones por medio de lo que comúnmente se denomina interfaz del objeto, que estará constituido por los métodos públicos.

Los datos y la implementación queda oculta a los demás objetos que interaccionan en el programa, lo que favorece enormemente la *protección de los datos* y las estructuras internas contra las modificaciones externas al objeto. De este modo es mucho más sencillo localizar errores en los programas puesto que cada objeto está altamente especializado, y sólo se encarga de su tarea. Como se puede observar, esto consigue una mayor *modularidad*, que facilita además el diseño en equipo de programas y la reutilización de clases (componentes) creados por otros desarrolladores.

Otro concepto importante en POO es la **abstracción**, indica la capacidad de *ignorar* determinados aspectos de la realidad con el fin de facilitar la realización de una tarea. Nos permite ignorar aquellos aspectos de la realidad que *no intervienen en el problema* que deseamos abordar, y también nos permite ignorar los *aspectos de implementación* de los objetos en los pasos *iniciales*, con lo cual sólo necesitamos conocer *qué* es lo que hace un objeto, y no cómo lo hace, para definir un objeto y establecer las relaciones de éste con otros objetos.

Un *objeto* lo podríamos representar como dos circunferencias, una interna que permanece oculta al mundo exterior y que contendría todos los detalles de la implementación del objeto, y otra circunferencia concéntrica externa, que representa lo que el objeto muestra al mundo exterior y le permite utilizar para interactuar con él.



La *encapsulación* ofrecida a través de objetos tiene varios beneficios, entre los que destacan la modularidad y la ocultación de la información.

Mediante la *modularidad* podemos escribir código de manera independiente de cómo se encuentren construidos los diferentes objetos que vamos a utilizar. Y ocultando la información se permite realizar cambios en el código interno de los objetos sin que afecte a otros objetos que los utilicen o dependan de ellos. No es necesario entender la implementación interna de un objeto para poder utilizarlo

2.2 Clase

Una clase es un molde o prototipo que define un tipo de objeto determinado. Una clase define los atributos y métodos que va a poseer un objeto. Mediante las clases podremos crear o instanciar objetos de un mismo tipo, estos objetos se distinguirán unos de otros a través de su estado, es decir, el valor de sus atributos.

La clase la vamos a utilizar para definir la estructura de un objeto, es decir, estado (atributos) y comportamiento (métodos). La clase es un concepto abstracto que generalmente no se va a utilizar directamente en nuestros programas o aplicaciones. Lo que vamos a utilizar van a ser objetos concretos que son instancias de una clase determinada.

La clase es algo genérico y abstracto, es similar a una idea. Cuando decimos piensa en un coche todos tenemos en mente la idea general de un coche, con puertas, ruedas, un volante, etc., sin embargo cuando decimos "ese coche que está aparcado ahí fuera", ya se trata de un coche determinado, con una matrícula, de un color, con un determinado número de puertas, y que podemos tocar y utilizar si es necesario. Sin embargo como ya hemos dicho la clase es la idea que define al objeto concreto.

Un ejemplo que se suele utilizar para diferenciar y relacionar clases y objetos es el ejemplo del molde de galletas. El molde para hacer galletas sería una clase, y las galletas que hacemos a partir de ese molde ya son objetos concretos creados a partir de las características definidas por el molde. Una vez implementada una clase podremos realizar instancias de la misma para crear objetos que pertenezcan a esa clase.

Las clases ofrecen el beneficio de la reutilización, utilizaremos la misma clase para crear distintos objetos. Y luego veremos que una vez que tenemos una clase podremos aprovecharla heredando de ella para complicarla o especializarla para una labor concreta.

Si comparamos las clases y objetos de la POO con la programación estructurada tradicional, se puede decir que las clases son los tipos de datos y los objetos las variables de esos tipos de datos. De esta forma si tenemos el tipo entero, en la POO diríamos que es la clase entero, y si tenemos una variable de tipo entero, en la POO diríamos que tenemos un objeto de la clase entero.

2.3 Mensajes

Los mensajes son la forma que tienen de comunicarse distintos objetos entre sí. Un objeto por sí sólo no es demasiado útil, sino que se suele utilizar dentro de una aplicación o programa que utiliza otros objetos.

El comportamiento de un objeto está reflejado en los mensajes a los que dicho objeto puede responder. Representan las acciones que un determinado objeto puede realizar.

Un mensaje enviado a un objeto representa la invocación de un determinado método sobre dicho objeto, es decir, la ejecución de una operación sobre el objeto. Es la manera en la que un objeto utiliza a otro, el modo en el que dos objetos se comunican, ya que la ejecución de ese método retornará el estado del objeto invocado o lo modificará.

Los mensajes se utilizan para que distintos objetos puedan interactuar entre sí y den lugar a una funcionalidad más compleja que la que ofrecen por separado. Un objeto lanzará o enviará un mensaje a otro objeto si necesita utilizar un método del segundo objeto. De esta forma si el objeto A quiere utilizar un método del objeto B, le enviará un mensaje al objeto A.

Para enviar un mensaje se necesitan tres elementos:

- el objeto al que se le va a enviar el mensaje.
- el nombre del método que se debe ejecutar.
- los parámetros necesarios para el método en cuestión.

2.4 Herencia

La herencia es un mecanismo mediante el cual podemos reutilizar clases ya definidas. Es decir, si tenemos una clase botón que define un tipo de objeto que se corresponde con un botón que tiene un texto, que se puede pulsar, etc., si queremos definir una nueva clase llamada botón de color, no tenemos que rescribir todo el código y crear una clase completamente nueva, sino lo que haremos será heredar de la clase botón, utilizar lo que nos ofrezca esta clase y añadirle lo que sea necesario para la nueva funcionalidad deseada.

La herencia dentro de la POO es un mecanismo fundamental que se puede definir también como una transmisión de las características de padres a hijos. Entendiendo aquí características como métodos y atributos de una clase. La clase hija puede añadir atributos, métodos y redefinir los métodos de la clase padre.

Podemos ver la herencia como una sucesiva especialización de las clases. La clase de la que se hereda se suele denominar *clase padre o superclase*, y la clase que hereda se denomina *clase hija o subclase*.

El mecanismo de herencia es muy potente, puesto que nos permite agregar funcionalidades nuevas a una clase ya existente, reutilizando todo el código que ya se tenga disponible de la clase padre, es decir, se heredarán sus atributos y métodos, como ya habíamos indicado con anterioridad. En las clases hijas podemos redefinir el comportamiento de la clase padre.

Mediante el mecanismo de herencia podemos definir superclases denominadas *clases abstractas* que definen comportamientos genéricos. De esta clase pueden heredar otras clases que ya implementan de forma más concreta estos comportamientos. De esta forma podremos crear jerarquías de clases.

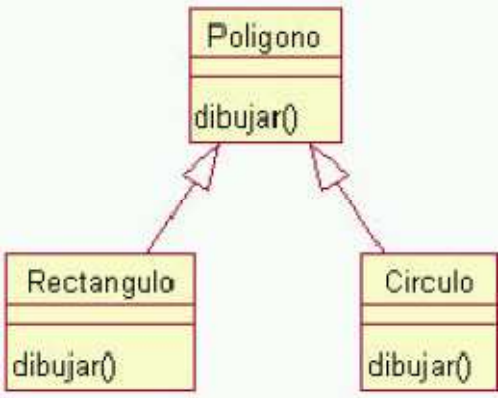
Los métodos que se heredan de la clase padre no tienen porqué utilizarse sin realizar ningún cambio, se puede llevar a cabo lo que se denomina la sobrescritura de métodos. Podemos heredar un método de la clase padre, pero en la clase hija le podemos dar una implementación diferente para que se adecue a la nueva clase.

2.5 Polimorfismo

Consiste en la posibilidad de tener métodos con el mismo nombre en distintas clases. Al hablar de métodos en distintas clases nos estamos refiriendo a métodos distintos y por tanto con comportamientos distintos a pesar de que tengan el mismo nombre.

El polimorfismo permite poder enviar un mismo mensaje (recordemos que un mensaje es una invocación a un método) a objetos de clases diferentes. Estos objetos recibirán el mismo mensaje pero responderán a él de formas diferentes. Por ejemplo, un mensaje "+" para un objeto entero significaría una suma, mientras que para un objeto String (cadena de caracteres) significaría la concatenación.

Revisando el siguiente gráfico:



En él podemos ver una jerarquía de clases en la que todas las clases que la componen tienen un método llamado `dibujar()`. Todos tienen el mismo nombre pero cada uno de ellos podrá tener una funcionalidad distinta. En este ejemplo concreto, una posible interpretación podría ser que tenemos dos clases hijas que redefinen el método `dibujar()` de su clase padre.

Probablemente el método `dibujar()` de la clase padre sea un método abstracto (no se ha implementado, sólo se ha definido) ya que para dibujar un polígono es necesario saber el tipo de polígono del que se trata. Por tanto, las clases hijas de la clase "Poligono" se ven obligadas a implementar el método `dibujar()` que será distinto en cada caso. Esto es un claro ejemplo de polimorfismo.

3. Preparando el entorno

En este punto vamos a ver un ejemplo teórico-práctico de la clase más sencilla que se puede realizar en Java. Para ello vamos a detallar punto por punto el contenido de la misma y vamos a dar una serie de "recomendaciones" que a partir de ahora deberíamos de seguir en la construcción de nuestras clases.

3.1. Mi primera clase

Aunque todavía no conocemos la sintaxis básica del lenguaje (la veremos en el siguiente punto), podemos echarle un vistazo a una clase básica en Java.

```
1     // Fichero HelloWorld.java
2     [public]class HelloWorld {
3           public static void main(String[] args) {
4                 System.out.println("Hello Word!");
5           }
6     }
```

NOTA: Los números que encabezan cada línea sólo son para hacer referencia a ellas en las explicaciones y no formarían parte del código.

Línea 1

Es un simple comentario de tipo línea en el que hemos colocado el nombre del fichero. El compilador ignorará todo lo que va desde los caracteres "//" hasta el final de la línea. Como esta línea no tiene mayores explicaciones, aprovechamos para hacer hincapié acerca del uso comedido de comentarios en los programas. Tan mala es la falta de comentarios como el abuso de ellos. Pensemos que los comentarios son de mucha utilidad, tanto para otras personas que tengan que revisar nuestro código, como para nosotros mismos en futuras revisiones.

Línea 2:

Declara el nombre de la clase. Usamos la palabra reservada **class** seguida del nombre que queremos darle a nuestra clase, en este caso es "HelloWorld". Ya sabemos que Java es un lenguaje orientado a objetos, por tanto, nuestro programa ha de ser definido como una clase. Por convención, las clases Java se definen con la primera letra en mayúsculas.

Línea 3:

Aparentemente es más compleja, pero veremos que no es así. Estamos ante una declaración de un método de la clase que estamos definiendo. Este método se llama **main**, se define como público (**public**), estático (**static**) y no va a devolver nada (**void**) cuando se ejecute.

Se define como público para que pueda ser conocido fuera de la clase y como estático para disponer de él sin tener que crear un objeto. Después del nombre, y encerrada entre paréntesis, vemos la definición de los parámetros que puede recibir el método. En nuestro ejemplo se trata de un parámetro de nombre "args" y de tipo array de cadenas de texto. Este método será el punto de partida inicial de nuestro programa cuando lo ejecutemos.

NOTA: A partir de la versión 1.5 de Java, nos podemos encontrar este método definido de la siguiente manera (lo explicaremos más adelante):

```
public static void main(String… args)
```

Línea 4:

Puede resultar extraña para aquellos que no conozcan la sintaxis de la programación orientada a objetos. Se trata de una llamada al método **println** del flujo de salida (**out**) estándar del sistema (**System**). Dicho método se encarga de mostrar por consola la línea que se le indique.

Las líneas restantes son el cierre de la definición del método **main** (línea 5) y el cierre de la definición de la clase **HelloWorld** (línea 6).

Ahora vamos a *compilar* nuestra clase. Para ello usaremos la herramienta "javac" indicándole el nombre de nuestra clase. A continuación vemos como hacerlo:

```
javac HelloWorld.java
```

Si lo hemos hecho todo bien no obtendremos ningún mensaje al terminar la compilación y habremos obtenido un fichero llamado "HelloWorld.class".

Lo siguiente es la ejecución. Veremos, por tanto, el resultado de nuestro trabajo. Ahora invocaremos al intérprete, llamado "java", indicándole qué queremos ejecutar.

```
java HelloWorld
```

3.2. Convenciones y nomenclatura en la programación Java

Con respecto a los nombres de variables, las reglas del lenguaje Java son muy amplias y permiten mucha libertad, pero es habitual seguir ciertas normas que faciliten la lectura y el mantenimiento de los programas. Ya hemos dicho que los nombres en Java son *sensibles a mayúsculas y minúsculas*. Por ejemplo, podemos tener en un programa las variables "altura", "Altura" y "ALTURA" y serían tres variables distintas.

Por convención, se recomienda seguir las siguientes reglas:

- Normalmente se emplean nombres con minúsculas salvo las excepciones que se enumeran en los siguientes puntos.
- En los nombres que se componen de varias palabras es aconsejable colocar una detrás de otra poniendo en mayúscula la primera letra de cada palabra.
- Los nombres de las clases y las interfaces empiezan siempre por mayúscula.
- Los nombres de objetos, métodos y variables empiezan siempre por minúscula.
- Los nombres de las variables finales (las constantes) se definen siempre con mayúsculas.

Veamos a continuación un ejemplo con las reglas anteriormente descritas:

```
// Es una clase
class Circunferencia {
	// Es una variable final
	private final double PI = 3.14159;

	// Es una variable miembro de la clase
	private double elRadio;

	// Son un método y una variable local
	public void establecerRadio(double radio) {
		elRadio = radio;
	}

	// Es un método
	public double calcularLongitud() {
		return (2 * PI * elRadio);
	}
}
```

4. Sintaxis del lenguaje

En este capítulo vamos a tratar la sintaxis del lenguaje Java. Si conocemos C o C++ veremos que la sintaxis es muy similar, de todas formas repasemos este capítulo. Este capítulo es necesario para saber cómo escribir el código Java, en el próximo capítulo veremos cómo implementar los mecanismos de la POO a través de Java. Veremos que muchos de los elementos del lenguaje Java son comunes al resto de los lenguajes existentes.

4.1. Identificadores

Los identificadores son literales que representan nombres únicos para ser asignados a objetos, variables, clases y métodos, de este modo el compilador puede identificarlos unívocamente. Estos nombres sirven al programador, si éste les da sentido.

Existen de todos modos algunas limitaciones a la hora de dar nombres a los identificadores, que son las mismas que en la mayoría de los lenguajes:

- Los identificadores tienen que comenzar por una letra, un subrayado (_) o un símbolo de dólar ($). *¡NO pueden comenzar con un número!*

- Tras el primer carácter, se pueden utilizar combinaciones de los ya mencionados y números del 0 al 9. No pueden utilizarse, como es lógico, las palabras clave del lenguaje como identificadores (que mostraremos más adelante). Aunque estos identificadores pueden ser cualquier longitud, para el compilador sólo son significativos los primeros 32 caracteres.

- Java distingue entre mayúsculas y minúsculas, de este modo, *Nombre* y *nombre* son dos variables diferentes.

- Java permite el uso de cualquier carácter del código Unicode para definir identificadores, de esta forma, el identificador *Año* es totalmente válido en Java.

LEGAL	ILEGAL

`int _a;`
`int $c;`
`int ______2_w;`
`int _$;`
`int`
`this_is_a_very_detailed_name_for_an_identifier;`
	`int :b;`
	`int -d;`
	`int e#;`
	`int .f;`
	`int 7g;`

4.2. Palabras clave

Las palabras clave son identificadores reservados por el lenguaje. Deberíamos leer con detenimiento la siguiente tabla, ya que Java tiene un número mayor de palabras reservadas que C y C++.

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

4.3. Comentarios

Los comentarios en java pueden hacerse de dos formas:

a) Comentarios de una línea

```
int variable; //Esto es un comentario al final de línea.
```

b) Comentario de varias líneas

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

Existe un tercer tipo de comentarios, utilizado por el sistema *javadoc* de documentación de Java. *Javadoc* permite crear documentación de nuestros programas en HTML con el mismo formato que el API de Java.

4.3.1 Formato JavaDoc

El tipo de comentario JavaDoc se escriben comenzando por /** y terminando con */ , pudiendo ocupar varias líneas. Mientras que los comentarios usuales no tienen ningún formato, los comentarios JavaDoc siguen una estructura prefijada.

Los comentarios JavaDoc están destinados a describir, principalmente, clases y métodos. Como están pensados para que otro programador los lea y utilice la clase (o método) correspondiente, se decidió fijar al menos parcialmente un formato común, de forma que los comentarios escritos por un programador resultaran legibles por otro. Para ello los comentarios JavaDoc deben incluir unos indicadores especiales, que comienzan siempre por '@' y se suelen colocar al comienzo de línea.

Por ejemplo:

```
/**
 * Una clase para representar círculos situados sobre el plano.
 * Cada círculo queda determinado por su radio junto con las
 * coordenadas de su  centro.
 * @version 1.2, 24/12/04
 * @author Rafa Caballero
 */

public class Círculo {
    protected double x,y; // coordenadas del centro
    protected double r;  // radio del círculo

    /**
     * Crea un círculo a partir de su origen su radio.
     * @param x La coordenada x del centro del círculo.
     * @param y La coordenada y del centro del círculo.
     * @param r El radio del círculo. Debe ser mayor o igual a 0.
     */
    public Circulo(double x, double y, double r) {
        this.x=x; this.y = y; this.r = r;
    }

    /**
     * Cálculo del área de este círculo.
     * @return El área (mayor o igual que 0) del círculo.
     */
     public double área() {
        return Math.PI*r*r;
     }

…
```

Como se ve, y esto es usual en JavaDoc, la descripción de la clase o del método no va precedida de ningún indicador. Se usan indicadores para el número de versión (@version), el autor (@author) y otros.

Es importante observar que los indicadores no son obligatorios; por ejemplo en un método sin parámetros no se incluye obviamente el indicador @param. También puede darse que un comentario incluya un indicador más de una vez, por ejemplo varios indicadores @param porque el método tiene varios parámetros. Vamos a hacer un resumen de los indicadores más usuales:

INDICADOR	DESCRIPCIÓN
@author nombreDelAutor descripción	Indica quién escribió el código al que se refiere el comentario. Si son varias personas se escriben los nombres separados por comas o se repite el indicador, según se prefiera. Es normal incluir este indicador en el comentario de la clase y no repetirlo para cada método, a no ser que algún método haya sido escrito por otra persona.
@version númeroVersión descripción	Si se quiere indicar la versión. Normalmente se usa para clases, pero en ocasiones también para métodos.
@param nombreParámetro descripción	Para describir un parámetro de un método.
@return descripción	Describe el valor de salida de un método.
@see nombre descripción	Cuando el trozo de código comentado se encuentra relacionada con otra clase o método, cuyo nombre se indica en *nombre*.
@throws nombreClaseExcepción descripción	Cuando un método puede lanzar una excepción ("romperse" si se da alguna circunstancia) se indica así.
@deprecated descripción	Indica que el método (es más raro encontrarlos para una clase) ya no se usa y se ha sustituido por otro

4.3.2 La herramienta JavaDoc

Hemos visto cuál es el formato de los comentarios JavaDoc. Aparte de obtenerse comentarios más fáciles de leer para otros programadores debido al formato impuesto por los indicadores, la principal utilidad de estos comentarios es que pueden utilizarse para generar la documentación de los programas. Para ello se utiliza la herramienta *javadoc* parte de JDSK. El formato más sencillo de esta herramienta, cuando se emplea desde línea de comandos es:

```
javadoc fichero.java
```

Lo que hace esta herramienta es extraer los comentarios JavaDoc contenidos en el programa Java indicado y construir con ellos ficheros .html que puede servir como documentación de la clase. Por ejemplo, esta es el fichero Circulo.html que genera javadoc (entre otros) al procesar el fichero Circulo.java:

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

figuras

Class Círculo

```
java.lang.Object
  └figuras.Círculo
```

```
public class Círculo
extends java.lang.Object
```

Una clase para representar círculos situados sobre el plano. Cada círculo queda determinado por su radio junto con las coordenadas de su centro.

Constructor Summary

Círculo(double x, double y, double r) Crea un círculo a partir de su origen su radio.

Method Summary

double	área() Cálculo del área de este círculo.
boolean	contiene(double px, double py) Indica si un punto está dentro del círculo.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Círculo

```
public Círculo(double x,
               double y,
               double r)
```

Crea un círculo a partir de su origen su radio.

4.4. Tipos de datos

Java es un lenguaje sencillo y con pocos tipos de datos, lo cual no implica que no sea potente, de hecho, es un lenguaje moderno con el que podemos crear prácticamente cualquier tipo de aplicación que se necesite.

Los tipos de datos se pueden clasificar en dos categorías:

- *Tipos primitivos*

  Son, al igual que en otros lenguajes, los tipos básicos y no constituyen objetos propiamente dichos. Entre ellos están los tipos byte, short, int, long, char, float, double y boolean. A su vez pueden clasificarse en tipos enteros (byte, short, int, long y char), tipos de coma flotante (float y double) y tipo lógico (bolean).

- *Tipos de referencia.*

  Se usan para hacer referencia a los objetos. Existen tres tipos:
  - las referencias de objeto.
  - las referencias de interfaz.
  - las matrices.

Para todos los tipos primitivos existe una clase java, dentro del paquete "java.lang", capaz de representar a cada uno de ellos como objeto. Estas clases reciben el nombre de "Wrappers" (o envoltorios). Por ejemplo, java.lang.Byte para el tipo byte, java.lang.Short para el tipo short, etc. Se irán viendo en las distintas tablas por cada tipo primitivo. Asimismo, todas esas clases tienen su correspondiente método "toString()", que devuelve el número convertido a texto.

Antes de empezar a explicar todos los tipos de datos existentes en Java vamos a hablar de un par de cosas que tenemos que tener en cuenta antes de empezar a programar en este lenguaje, vamos a ver en qué consiste el recolector de basura "*Garbage Collector*", y que significa el almacenamiento en Stack o Heap.

4.4.1 El recolector de basura "Garbage Collector"

En muchos lenguajes orientados a objetos es necesario seguir la pista de los objetos que se vayan creando para luego destruirlos cuando no sean necesarios. El código necesario para gestionar así la memoria es tedioso y propenso a errores. Podemos encontrar algunos programas que mientras se están ejecutando van acaparando memoria que no llegan a liberar después. En algunos casos, incluso, se hace necesario reiniciar el sistema para liberar la memoria que había quedado permanentemente ocupada.

Java nos facilita el trabajo. Nos permite crear tantos objetos como necesitemos y nos libera de la tediosa tarea de destruirlos cuando no sigan siendo necesarios. El entorno de ejecución de Java elimina los objetos cuando determina que no se van a utilizar más, es decir, que han dejado de ser necesarios. Este proceso es conocido como *recolección de basura*.

Un objeto puede ser elegido por el recolector de basura cuando no existen más referencias a él. Existen dos motivos por los cuales un objeto deja de estar referenciado:

- La variable que mantiene la referencia salga de su ámbito.
- Se borra explícitamente un objeto referencia mediante la selección de un valor cuyo tipo de dato es una referencia a *null*.

El entorno de ejecución de Java tiene un recolector de basura que periódicamente libera la memoria ocupada por los objetos que no se van a necesitar más. El recolector de basura de Java es un proceso que dinámicamente examina la memoria en busca de objetos. Marca todos aquellos que están siendo referenciados y cuando termina, los que no están marcados los elimina.

El recolector de basura funciona en un *thread* (hilo de ejecución) de baja prioridad y funciona tanto síncrona como asíncronamente, dependiendo de la situación y del sistema en el que se esté ejecutando el entorno

Java se ejecuta *síncronamente* cuando el sistema funciona fuera de memoria o en respuesta a una petición de un programa Java que lo solicite mediante una llamada a `System.gc()`. Es importante que tengamos en cuenta que cuando solicitamos que se ejecute el recolector de basura no tenemos garantías de que los objetos vayan a ser recolectados inmediatamente.

En aquellos sistemas que permiten que el entorno de ejecución Java sepa cuando un thread ha empezado a interrumpir a otro thread, el recolector de basura de Java funciona ***asíncromamente*** cuando el sistema está ocupado. Tan pronto como otro thread se vuelva activo, se pedirá al recolector de basura que obtenga un estado consistente y termine.

Analicemos el siguiente código:

```
1     class MyClass {
2           public static void main(String[] args) {
3                 MyClass myFirstObject = new MyClass();
4                 MyClass mySecondObject = new MyClass();
5                 myFirstObject = mySecondObject;
6                 mySecondObject = null;
7           }
8     }
```

¿Qué está ocurriendo?

- *Línea 3*-Se crea el primer objeto en el heap, se crea la primera referencia en el stack, y la referencia apunta al objeto.
- *Línea 4*-Se crea el segundo objeto en el heap, se crea la segunda referencia en el stack, y la referencia apunta al objeto.
- *Línea 5*-La primera referencia apunta hacia el segundo objeto. En este momento el primer objeto se queda sin referencia, es desechado por el GC, y se libera memoria.
- *Línea 6*-La segunda referencia, apunta a nada.

Otro código más:

```
1     class MyClass {
2           public MyClass myProperty;
3           public static void main(String[] args) {
4                 MyClass myFirstObject = new MyClass();
5                 myFirstObject.myProperty = new MyClass();
6                 myFirstObject = null;
7           }
8     }
```

- *Línea 4*-Creamos un objeto de la clase MyClass, creamos la referencia….
- *Línea 5*-myProperty de `myFirstObject` empieza a apuntar a un nuevo objeto de la clase MyClass, y ahora tenemos dos objetos, `myFirstObject`, y el nuevo que esta referenciado por `myFirstObject`, el atributo `myProperty`.
- *Línea 6*-Ahora la referencia `myFirstObject`, deja de apuntar al primer objeto creado y ahora apunta a *null*(es decir a ninguna parte). Ahora, ¿Cuántos objetos tenemos? Ninguno, porque al eliminarse el objeto `myFirstObject`, se elimina (en 'cascada') el objeto al que apuntaba myProperty (atributo de `myFirstObject`).

Finalización de los objetos

Justo antes de que el objeto sea eliminado definitivamente, el recolector de basura le da una oportunidad para limpiarse él mismo mediante la llamada al método `finalize()` del propio objeto. Este proceso es conocido como *finalización*.

Durante el proceso de finalización, un objeto podría liberar los recursos que tenga en uso, como pueden ser los ficheros abiertos, conexiones a bases de datos, registros de auditoría, etc. y así poder liberar referencias a otros objetos para que puedan ser seleccionados también por el recolector.

El método `finalize()` es un miembro de la clase `java.lang.Object` y cualquier clase Java que lo necesite debe sobrescribirlo e indicar en él las acciones necesarias a realizar cuando se eliminen los objetos de ese tipo.

Veamos un ejemplo:

```
1     public class MyClass {
2           String name;
3           MyClass(String name){
4                 this.name=name;
5           }
6           protected void finalize(){
7                 System.out.println(this.name);
8           }
9           public static void main(String[] args) {
10                MyClass myFirstObject=new MyClass("Ed");
11                MyClass mySecondObject=new MyClass("Edd");
12                MyClass myThirdObject=new MyClass("Eddy");
13                myFirstObject=null;
14                mySecondObject=null;
15                myThirdObject=null;
16                System.gc();
17          }
18    }
```

Se crean tres objetos con su respectiva referencia, luego se les da la referencia a *null*, y finalmente le sugerimos al GC que pase. ¿Qué saldría por pantalla?

```
C:\Documents and Settings\Administrador\Escritorio\raiz>java EjemploFinalize
Eddy
Edd
Ed
```

Se borra primero el último objeto que se creó y que ya no tiene referencia.

4.4.2 Stack y Heap

Una computadora, dentro de su memoria RAM, tiene dos divisiones, conocidas como "heap" y "stack", o "pila" y "cola". La diferencia entre ambas es la forma en la que se accede a éstas; para efectos prácticos, esto no nos interesa porque ese acceso lo hace el sistema operativo.

Stack

El **Stack**, para explicarlo de una manera más rápida, podríamos imaginarlo como si fuese un *Array*, que es de tamaño fijo durante el tiempo ejecución del programa, en tiempo de compilación es cuando se define el tamaño que tendrá.

En concreto se guardan dos tipos de elementos:

- las referencias a objetos (o instancias).
- datos de tipo primitivo (int, float, char…), se almacenan las variables locales, parámetros/valores de retorno de los métodos, etc.

Por ejemplo, si ejecutamos un método, las variables locales de ese método se guardarían en el Stack, y al finalizar el método, se borran del Stack, pero hay que tener en cuenta que el Stack es fijo, y solamente estaríamos usando y dejando de usar una parte del Stack, y por esa misma característica de ser estático, si lo llenamos, caeríamos en un *StackOverflowError*.

Heap

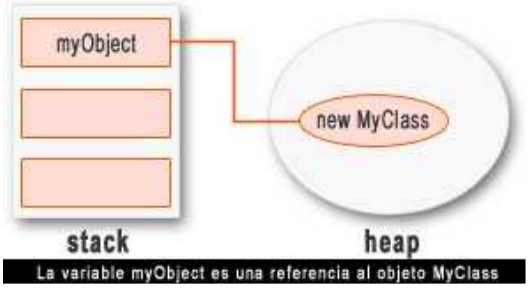
El Heap**,** esta es la zona de la memoria dinámica, los objetos son creados, eliminados o modificados en esta parte de la memoria, la Java Virtual Machine (desde ahora, la JVM), le asigna un espacio predeterminado de memoria, pero al necesitar más memoria, la JVM le solicita al sistema operativo la memoria necesitada, y según se vaya requiriendo, la JVM le seguirá proporcionando más memoria.

Cómo interactúan el Stack y el Heap

Como ya hemos dicho el stack guarda referencias y datos primitivos, las referencias pueden apuntar a elementos del heap(es decir, a los objetos), las referencias tienen un tipo, que es al tipo de objeto que pueden apuntar las referencias, viéndolo de otra manera, que está determinado por la clase de la que se creará a la instancia, y puede apuntar a esa clase o subclases(elementos que hereden de la clase), el stack es fijo, y los tipos de referencias no pueden cambiar.

Por ejemplo:

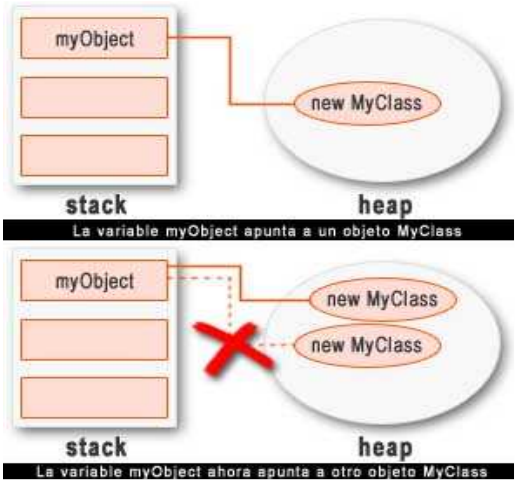
```
1  class MyClass {
2    public static void main(String[] args){
3      MyClass myObject;
4      myObject = new MyClass();
5    }
6  }
```



En la tercera línea, se crea la referencia en el stack, en la cuarta línea se crea el objeto en el heap y en el mismo momento hacemos la referencia desde el stack. Gráficamente esta sería una manera de representarlo:

Otro código que tenemos que estudiar es el siguiente:

```
1  class MyClass {
2    public static void main(String[] args){
3      MyClass myObject;
4      myObject = new MyClass();
5      myObject = new MyClass();
6    }
7  }
```



Algo que sería bueno comentar, es que la relación *referencia-objeto* es de muchos a uno, es decir, un objeto puede estar siendo apuntado por muchas referencias, pero una referencia solo apunta a un objeto.

4.4.3 Tipo primitivo entero

Como indica su propio nombre, se usan para almacenar números enteros. Las diferencias entre ellos son los rangos de valores soportados. El tipo *byte*, por ejemplo, es un entero de 8 bits con signo en complemento a dos, por lo tanto es capaz de almacenar valores comprendidos entre –128 y 127. Por otra parte, el tipo short es un entero de 16 bits con signo y en complemento a dos con un rango que va de – 32.768 a 32.767.

Todos los tipos enteros tienen signo y se representan en complemento a dos, excepto el tipo char, que es un entero sin signo de 16 bits y se usa para representar caracteres Unicode.

TIPO	TAMAÑO(bits)	RANGO	WRAPPER
byte	8	-128 a 127	`java.lang.Byte`
short	16	-32.768 a 32.767	`java.lang.Short`
int	32	-2.147.483.648 a 2.147.483.647	`java.lang.Integer`
long	64	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	`java.lang.Long`
char	16	0 a 65.535	`java.lang.Character`

4.4.4 Tipo primitivo coma flotante

Existen sólo dos tipos: float y double, y al igual que ocurría con los enteros, la diferencia está en los rangos de valores que pueden almacenar. El tipo float es de 32 bits de precisión sencilla, mientras que el tipo double es de 64 bits y de doble precisión

TIPO	TAMAÑO (bits)	VALOR MAX	VALOR MIN	WRAPPER
float	32	+-3.4028236e+38	+-1.4e-45	`java.lang.Float`
double	64	+-1.7984828356536153e+308	+-4.9e-324	`java.lang.Double`

4.4.5 Tipo primitivo boolean

Sólo existe un tipo y puede tomar los valores “true” (verdadero) y “false” (falso). Al contrario que en otros lenguajes, por ejemplo en C, no podemos usar los valores enteros en expresiones que requieran valores booleanos.

4.4.6 Secuencias de escape

Hay determinados caracteres en Java que o bien no tienen una representación explicita o bien no puede ser utilizados directamente por el hecho de tener un significado espacial para el lenguaje. Para poder utilizar estos caracteres dentro de un programa Java se utilizan las secuencias de escape.

Una secuencia de escape está formada por el carácter "\" seguido de una letra, en el caso de ciertos caracteres no imprimibles, o del carácter especial. La tabla siguiente contiene las principales secuencias de escape predefinidas en Java.

SECUENCIA DE ESCAPE	SIGNIFICADO
\b	Retroceso
\n	Salto de línea
\t	Tabulación horizontal
****	Barra invertida \
\'	Comilla simple
\"	Comilla doble

Por ejemplo para mostrar la siguiente salida:

```
C:\Documents and Settings\Administrador\Escritorio\raiz>java Secuencias
Hola            "Carlos"
pasa buen 'dia'
```

```
System.out.println("Hola \t\t\t\"Carlos\"\npasa buen \'dia\'");
```

4.4.7 Literales

Un literal es un valor constante que se puede asignar directamente a una variable o pueden ser utilizado en una expresión.

Existen cuatro tipos de literales básicos, coincidiendo con los cuatro grupos en los que se puede dividir los tipos básicos de Java, esto es, *numéricos enteros, numéricos decimales, lógicos y carácter.* A la hora de utilizar estos literales en una expresión hay que tener en cuenta lo siguiente:

- **Los literales numéricos enteros se consideran de tipo int.** Dado que todo literal entero es un tipo int, una operación como:

  ```
  byte b = 10;
  ```

  intentaría asignar un número de tipo int a una variable de tipo byte, lo podría parecer un error de compilación. Sin embargo, para este tipo de expresiones Java realiza una conversión implícita del dato al tipo destino siempre y cuando el dato "quepa" en la variable.

- **Los literales numéricos decimales se consideran de tipo double.** Así una asignación del tipo:

```
float p = 3.14;
```

provoca un error de compilación al intentar asignar un dato doublé a una variable float que tiene un tamaño menor. Para evitar el error, debemos de utilizar la letra "f" a continuación del número:

```
float p = 3.14f; //lo que provoca una conversión
```

- **Los literales boolean son *true* y *false*.** Estas palabras reservadas no tienen equivalencia numérica, por lo que la siguiente instrucción provocará un error de compilación de incompatibilidad de tipos:

```
boolean b = 0;
```

- **Los literales de tipo char se escriben entre comillas simples.** Se pueden utilizar la representación carácter o su valor Unicode en hexadecimal, precedido de la secuencia escape \u:

```
char car = '#';
char p = '\u03AF';
```

dado que el carácter es realmente un número entero, también puede asignarse directamente a una variable char el literal entero correspondiente a la combinación del carácter:

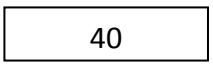
```
char c = 231; //almacena el carácter con código Unicode 231
```

4.4.8 Variables

Una variable es un espacio físico de memoria donde un programa puede almacenar un dato para su posterior utilización.

Los diferentes tipos de variables los acabamos de ver, por ejemplo, si hiciéramos:

```
int variableEntera = 25;
```



Los objetos en Java también se tratan a través de variables, sólo que, a diferencia de los tipos primitivos, una variable de tipo objeto no contiene al objeto como tal sino una referencia al mismo:

```
Clase cl = new Clase();
```

Declaración de una variable

Fijándonos un poco en los ejemplos anteriores podemos deducir que una variable se declara de la siguiente forma:

```
tipo_dato    nombre_variable
```

También podemos declarar en una misma instrucción varias variables de igual tipo:

```
tipo_dato    variable1, variable2;
```

Asignación

Una vez declarada la variable se le pude asignar un valor siguiendo el siguiente formato:

```
variable = expresión;
```

donde expresión puede ser cualquier expresión Java que devuelva un valor acorde al tipo de dato de la variable.

Algunos ejemplos.

```
int p, k, j;

p = 30;

k = p + 20;

j = k + p;
```

Ámbito de las variables

Según donde este declara la variable, esta puede ser:

- ***Variables de clase o miembro o atributo***: se les llama así a las variables que se declaran al principio de una clase, y fuera de los métodos. Estas variables son compartidas por todos los métodos de la clase y, deberían estar declaradas como *private* para limitar su uso al interior de la clase.
  Este tipo de variable puede ser utilizada sin haberlas inicializado explícitamente, ya que se inicializan implícitamente cuando se crea el objeto de la clase. A continuación, y dependiendo del tipo de la variable vemos su valor por defecto:

TYPO VARIABLE	VALOR POR DEFECTO
Referencias	null
byte, short, int, long	0
float, double	0.0
boolean	false
char	'\u000'

- ***Variables locales***: son variables que se declaran dentro de un método, su ámbito de utilización está restringido al interior del método y no admiten ningún tipo de modificador. Se crean en el momento en el que se hace la llamada al método, destruyéndose cuando finaliza la ejecución de éste.
  Una variable local también puede estar declarada dentro de un bloque de instrucciones, sólo que, en este caso su uso está restringido al interior de ese bloque. **Toda variable local tiene que ser inicializada explícitamente antes de ser utilizada.**

Por ejemplo:

```
1     class Variables{
2           private int variableEntera;
3           public void metodo(){
4                 int variableLocal = 0;
5                 if(...){
6                       int variableBloque = 0;
7                 }
8           }
9     }
```

4.5. Conversiones

Java es un lenguaje fuertemente tipado, lo que significa que es bastante estricto a la hora de asignar valores a las variables. A priori, el compilador sólo admite asignar a una variable un dato del tipo declarado en la variable, no obstante, en ciertas circunstancias, es posible realizar conversiones que permitan almacenar en una variable un dato de un tipo diferente al declarado.

En Java es posible realizar conversiones entre todos los tipos básicos, con excepciones de los boolean, que son incompatibles con el resto de los tipos.

Las conversiones de tipo pueden realizarse de dos maneras: implícitamente o explícitamente.

A partir de java 1.5, se admite otro tipo de conversión que facilita mucho el trabajo a los programadores, entre tipos primitivos y sus clases envolvente, lo que se conoce como el Autoboxing y el Unboxing, que explicaremos en el siguiente tema.

4.5.1 Conversiones implícitas

Este tipo de conversión se realiza de manera automática, es decir, el valor o expresión que va a asignar a la variable es convertido automáticamente al tipo de esta por el compilador, antes de almacenarlo en la variable. Veámoslo con un ejemplo:

```
int i;
byte b = 30;
i = b;
```

En el ejemplo, podemos ver como el dato de tipo *byte* almacenado en la variable b es convertido a *int* antes de ser asignado a la variable i.

Para que una conversión pueda realizarse de forma automática (implícitamente), **el tipo de la variable destino debe ser de tamaño igual o superior al tipo de origen**, si bien tenemos dos excepciones para esta regla:

- Cuando una variable es entera y el origen es decimal (*float* o *double*), la conversión no podrá ser automática.
- Cuando la variable destino es *char* y el origen es numérico, independientemente del tipo específico, la conversión no podrá ser automática.

Ejemplos de conversiones correctas:

```
int k = 5,p;            p = c; //conversión impl. char a int
short s = 10;           h = k; //conversion impl. int a float
char c = 'ñ';           k = s; //conversion impl. short a int
float h;
```

Ejemplos de conversiones incorrectas:

```
int n;            n = c; //error long a int no se puede
long c = 20;      k = s; //error byte a char no se puede
char k;           n = ft; //error float a int no se puede
float ft = 2.4f;
byte s = 4;
```

4.5.2 Conversiones explícitas

Cuando no se cumplen las condiciones para una conversión implícita, esta podrá realizarse explícitamente utilizando la siguiente expresión:

```
variable_destino = (tipo_destino) dato_origen;
```

A esta operación se le conoce como *casting* o estrechamiento ya que al convertir un tipo en otro de tamaño inferior se realiza una posible pérdida de información, aunque no conlleve errores en la ejecución.

Algunos ejemplos de conversión explicita:

```
char c;
byte k;
int p = 400;
double d = 34,6;

c = (char)d; //eliminamos la parte decimal (truncado)
k = (byte)p; //perdida de datos pero sin errores
```

4.6. Constantes

Una constante es una variable cuyo valor no pude ser modificado. Para definir una constante en Java se utiliza la palabra reservada *final*, delante de la declaración de tipo:

```
final tipo nombre_cte = valor;    final double pi = 3.1416;
```

Una constate se define en los mismos lugares en los que se declara una variable: al principio de la clase y en el interior de un método. Suele ser bastante habitual determinar constantes para que puedan ser utilizadas desde el exterior de la clase donde se han declarado, además de hacer que no sea necesario crear objetos para utilizar la constante, sería:

```
public static final double PI = 3.1416;
```

4.7. Expresiones

Una expresión no es más que una secuencia de elementos que puede ser evaluada por el compilador. Una llamada a una función, una asignación, un cálculo aritmético o lógico son algunos ejemplos de expresiones.

Las expresiones suelen dividirse según su tipo, es decir según el tipo del valor que devuelven al ser resueltas, en: *aritméticas* y *lógicas*. Dentro de cada una de ellas existen subdivisiones. El compilador las analiza y, si está bien construido, las simplifica todo lo posible, evaluándose en tiempo de ejecución.

Algunos ejemplos:

```
System.out.println( "Hola" + c );
a = 21;
b = a + b + c + 15;
c = (a + b) * (y-7);
d = 2 / 3 + 4 * 5;
d = a && b;
e = (a && b) || c;
```

Las expresiones se analizan de *izquierda a derecha*, salvo que existan *paréntesis*, en tal caso (como ocurre en matemáticas), los fragmentos de la expresión encerrados entre paréntesis son evaluados antes. Si no hubiese paréntesis, hay algunos fragmentos que son evaluados antes que otros, dependiendo de la precedencia (importancia) de los operadores que se encuentran entre los operandos, pero a igual nivel de precedencia, se mantiene la evaluación de *izquierda a derecha*.

4.8. Operadores

Son símbolos que se utilizan en algunas expresiones para modificar el valor de expresiones más simples, o bien, para crear expresiones nuevas a partir de otras. Como ejemplo podemos citar el operador de concatenación de cadenas, que se representa por un signo más (+) y que se usa para concatenar dos cadenas de texto en una nueva.

Los operadores no son genéricos, sino que se podrán usar dependiendo de las expresiones sobre las que se quiere operar. Como ejemplo podemos volver a citar el operador de concatenación de cadenas y el operador de suma de enteros. Aunque ambos se representen con el mismo símbolo (el más de la suma) no es el mismo operador.

4.8.1 Operadores sobre enteros

Estos operadores pueden aplicarse a todos los tipos de datos enteros: byte, short, int, long. Se dividen en:

Operadores Aritméticos para Números Enteros		
	Op.	**Descripción**
Unarios	**++**	Incremento. Añade una unidad al valor actual de la variable. (1)
	--	Decremento. Sustrae una unidad al valor actual de la variable. (1)
	-	Negación. Equivale a multiplicar por –1 el valor de la variable.
	~	Complemento a nivel de bits. (2)
Binarios	**+**	Adición
	-	Sustracción
	*****	Multiplicación
	/	División
	%	Módulo
	<<	Desplazamiento a la izquierda. (3)
	>>	Desplazamiento a la derecha. (3)
	>>>	Desplazamiento a la derecha con relleno de ceros. (3)
	&	AND a nivel de bits.
	|	OR a nivel de bits.
	^	XOR a nivel de bits.
Relacionales	**<**	Menor que
	>	Mayor que
	<=	Menor o igual que
	>=	Mayor o igual que
	= =	Igual que
	! =	Distinto que

Tanto los operadores unarios como los binarios, independientemente del tipo de los operandos sobre los que se realice la operación, devuelven un *int* para todos los casos, excepto que uno de los operandos sea un *long*, en cuyo caso el valor devuelto es de tipo *long*.

(1) Estos operadores pueden utilizarse en forma de prefijo (++*variable*) o forma de sufijo (*variable*++). En el primer caso el incremento (o decremento) de la variable se realiza antes de evaluar la expresión, y en segundo se realiza tras evaluar la expresión.

(2) Este operador conmuta el número a nivel de bits, es decir, todos los bits que estaban a 0 pasan a ser 1, y todos los bits que estaban a 1 pasan a 0.

(3) Estos operadores desplazan los bits a la derecha o izquierda el número de posiciones especificado como segundo operando.

4.8.2 Operadores sobre reales

Estos operadores trabajan sobre número de coma flotante, es decir, los tipos: *float* y *double*. Lo dicho anteriormente acerca de los operadores sobre enteros es aplicable a los operadores sobre números reales salvo que el resultado de la expresión será de tipo *float* si ambos operandos son de este tipo y en caso contrario el tipo devuelto será *double*.

Operadores Aritméticos para Números Reales		
	Op.	**Descripción**
Unarios	**++**	Incremento. Añade una unidad al valor actual de la variable.
	--	Decremento. Sustrae una unidad al valor actual de la variable.
Binarios	**-**	Adición
	-	Sustracción
	*****	Multiplicación
	/	División
	%	Módulo
Relacionales	**<**	Menor que
	>	Mayor que
	<=	Menor o igual que
	>=	Mayor o igual que
	= =	Igual que
	! =	Distinto que

4.8.3 Booleanos

Operador	**Descripción**
&	AND
|	OR
^	XOR
&&	AND (cortocircuito)
||	OR (cortocircuito)
!	NOT (negación)
==	Igualdad
!=	Distinto a
?:	Condicional

Son aquellos que efectúan operaciones sobre datos de tipo booleano, y como es lógico, el tipo de dato que devuelven es booleano.

4.8.4 Asignación

Operador	Descripción	Equivalencia
=	Simple	
+=	Adicción	var1 = var1 + var2 → var1 += var2
-=	Sustracción	var1 = var1 – var2 → var1 –= var2
*=	Multiplicación	var1 = var1 * var2 → var1 *= var2
/=	División	var1 = var1 \ var2 → var1 \= var2
%=	Módulo	var1 = var1 % var2 → var1 %= var2
&=	AND	var1 = var1 & var2 → var1 &= var2
|	OR	var1 = var1 | var2 → var1 |= var2
^=	XOR	var1 = var1 ^ var2 → var1 ^= var2

4.8.5 Procedencia de los operadores

A continuación se presenta todos los operadores que hemos visto, según su orden de precedencia de mayor a menor, es decir, según su importancia a la hora de ser evaluados.

Esto quiere decir, que salvo que se agrupen expresiones mediante el uso de paréntesis, aquellos operadores que tengan mayor orden de precedencia, serán evaluados primero.

- [] ()
- ++ --
- ! ~ instanceof
- / %
- + -
- << >> >>>
- < > <= >= = = !=
- & ^ |
- && ||
- ? :
- = += –= *= /= %= &= |= ^=

4.9. Instrucciones de control

Las sentencias de control de flujo nos permiten que ciertas partes de un programa no se ejecuten (condicionales) o que se ejecuten más de una vez (bucles). Sin ellas, los programas Java se ejecutarían de arriba abajo, se procesarían todas sus líneas y una sola vez cada una de ellas.

Las condiciones permiten bifurcar el flujo de la aplicación y en Java se dispone de las sentencias *ifelse* y *switch*, los bucles permiten repetir un fragmento de código un número determinado de veces, y las sentencias disponibles en Java para realiza bucles son: *for*, *while* y *do-while*.

4.9.1. if-else

El formato básico de una sentencia *if* es la siguiente:

```
if (<ExprBooleana>)
      sentencia1;
[else [if (<ExprBooleana>) ]
      sentencia2;]
```

Aunque también pueden utilizarse bloques en lugar de sentencias, del siguiente modo:

```
if (<ExprBooleana>) {
      sentencias;
}
[else [if (<ExprBooleana>) ] {
      sentencias;
}]
```

<ExprBooleana> puede ser cualquier expresión que, al ser evaluada, devuelva un resultado de tipo Booleano.

```
class ControlFlujo{
      public static void main(String args[]){
            String cadena1="hola";
            String cadena2="hola";
            String cadena3;
            if(cadena1.equals(cadena2)){
                  System.out.println("Iguales");
                  cadena3 = cadena2;
                  System.out.println("Cadena3: "+cadena3);
            }
            else
                  System.out.println("Diferentes");
      }
}
```

4.9.2. switch

La sintaxis de esta estructura es la siguiente:

```
switch( <Expresión> ) {
      case <Constante1>:
            sentencias;
            [break;]
      [case <Constante2>:
            sentencias;
            [break;]
      ......................
      ]
      [default:
            sentencias;
            [break;]]
}
```

En este caso `<Expresión>` debe de ser evaluado como un *char, byte, short, int* (o a partir de Java 1.5 una enumeración).

Algunos ejemplo:

```
byte g = 2;
switch(g) {
      case 23:
      case 128:
}
```

¿Tendremos algún problema con este código?

```
int temp = 90;
switch(temp) {
      case 80 : System.out.println("80");
      case 80 : System.out.println("80");
      case 90 : System.out.println("90");
      default :
      System.out.println("default");

}
```

¿Tendremos algún problema con este código?

```
switch(x) {
      case 0 {
            y = 7;
      }
}

switch(x) {
      0: { }
      1: { }

}
```

¿Tendremos algún problema con este código?

```
int x = 1;
switch(x) {
      case 1: System.out.println("1");
      case 2: System.out.println("2");
      case 3: System.out.println("3");
}
System.out.println("fuera");
```

¿Tendremos algún problema con este código?

```
int x = 7;
switch (x) {
      case 2: System.out.println("2");
      default: System.out.println("DD");
      case 3: System.out.println("3");
      case 4: System.out.println("4");
}
```

¿Tendremos algún problema con este código?

4.9.3. for

La sintaxis del bucle *for* es la siguiente:

```
for ( <ExprInicializa>; <ExprCondición>; <ExprIteración>){
      sentencias;
}
```

La ejecución de un bucle *for* comienza con la instrucción de inicialización, que suele realizar una inicialización de una variable de control (incluyendo su declaración). A continuación, se comprueba la condición cuyo resultado debe ser siempre de tipo *boolean*; en el caso de que sea *true* se ejecutarán las instrucciones delimitadas por el bloque {} y después se ejecutara la instrucción de incremento y volverá a comprobarse la condición.

Sobre la utilización de la instrucción *for* hay que tener en cuenta lo siguiente:

- Las instrucciones de control del bucle *for* (inicialización, condición e incremento) son opcionales. En cualquier caso el delimitar de instrucciones ";" es obligatorio.
- Si se declara una variable en la instrucción de inicialización, ésta será accesible únicamente desde el interior del *for.*
- Al igual que sucede con el *if*, las llaves delimitadoras de bloque solamente son obligatorias si el *for* está compuesto por más de una instrucción.

`for (int x = 1; x < 2; x++) {`
`  System.out.println(x);`
`}`
`System.out.println(x);`
	¿Tendremos algún problema con este código?

`for (int x = 10, y = 3; y > 3; y++) { }`	*¿Tendremos algún problema con este código?*

`for (int x = 0; ((((x < 10) && (y-- > 2)) | x == 3)); x++) { }`	*(a)*
`for (int x = 0; (x > 5), (y < 2); x++) { }`	*(b)*

`int b = 3;`
`for (int a = 1; b != 1;System.out.println("iterate")) {`
`  b = b - a;`
`}`
	¿Tendremos algún problema con este código?

Bucle for mejorado

En la versión 5 de Java se ha incluido una nueva sintaxis para el bucle *for* con la intención de hacer más legible el código. Se le conoce como bucle "for" mejorado (en inglés, *enhanced for*).

Sintaxis:

```
for (type var : arr) {body-of-loop}
for (type var : coll) {body-of-loop}
```

Lo pondremos en práctica y lo usaremos en más detalle cuando veamos los arrays.

4.9.4. while

Las principales diferencias entre el bucle *for* y el bucle *while* son que en éste no se inicializan los valores de control ni se dispone de la expresión de iteración.

La sintaxis es la siguiente:

```
while (CondBooleana){
      sentencias;
}
```

La sentencia o sentencias definidas en el cuerpo de un bucle *while* se repiten mientras que la condición de su cabecera sea verdadera. Es por esto que esta condición tiene que ser una expresión cuyo resultado sea de tipo booleano.

4.9.5. do-while

La sintaxis de este bucle es la siguiente:

```
do {
      sentencias;
}
while (CondBooleana);
```

Como puede verse, este bucle es casi idéntico al bucle *while*, salvo que en este caso, la expresión booleana se evalúa tras haberse realizado la primera iteración del bucle.

`int i = 0;`
`while(i < 10){`
`  System.out.println("Hola: "+i);`
`  i++;`
`}`
 	
`int j = 0;`
`do{`
`  System.out.println("Hola: "+j);`
`  j++;`
`}while(j < 10);`
	¿Tendremos algún problema con este código?

`int i = 0;`
`while(i = 2){`
`  System.out.println("Hola: "+i);`
`  i++;`
`}`
	¿Tendremos algún problema con este código?

4.9.6. Salida forzada de un bucle

Las instrucciones repetitivas *for* y *while*, cuentan con dos instrucciones que permiten abandonar la ejecución del bloque de instrucciones antes de su finalización. Estas instrucciones son: break y continue.

break

Ya hemos visto su utilidad dentro de un *switch*, sin embargo, su uso también se puede extender a las instrucciones repetitivas. En éstas, la utilización de break provoca una salida forzada del bucle continuando la ejecución del programa en la primera sentencia situada después del mismo.

Por ejemplo:

```
int numero = 6;
boolean acierto = false;
for(int i=1;i<5;i++){
	int aleat = (int)Math.floor(Math.random()*10+1);
	if(aleat == numero){
		acierto = true;
		break;
	}
}
```

continue

La instrucción *continue* provoca que el bucle detenga la iteración actual y pasa, en caso del *for*, a ejecutar la instrucción de incremento o, en caso del *while*, a comprobar la condición de entrada.

Por ejemplo:

```
int suma = 0;
while(suma < 100){
	int aleat = (int)Math.floor(Math.random()*10+1);
	if(aleat % 2 == 0){
		continue;
	}
	suma += aleat;
}
```

etiquetas

La instrucción *break* siempre hace que se termine inmediatamente un bucle y que la ejecución pase a la primera instrucción que siga al bucle. Cuando se usa con una *etiqueta*, el bucle también termina inmediatamente, pero la ejecución continúa en la primera instrucción que siga a la instrucción etiquetada.

La instrucción etiquetada puede ser un bucle externo (un bucle que encierra completamente al bucle en el que aparece la instrucción break con etiqueta) o un bloque de instrucciones en el que esté contenido el bucle.

En el siguiente ejemplo:

```
for(int i=1;i<=3;i++){
      for(int j=1;j<3;j++){
            if(j == 2){
                  break;
            }
            System.out.println(i+"x"+j+"="+(i*j));
      }
}
```

¿Cuál sería la salida?

```
fuera:
for(int i=1;i<=3;i++){
      for(int j=1;j<3;j++){
            if(j == 2){
                  break fuera;
            }
            System.out.println(i+"x"+j+"="+(i*j));
      }
}
```

¿Cuál sería la salida?

NOTA: También se permite el uso de etiquetas con *continue*;

NOTA: Cuando la instrucción *break* con etiqueta aparece dentro de una instrucción *try* que tiene asociado un bloque *finally*, al ejecutarse dicha instrucción *break* se ejecuta primero el bloque *finally*, antes de que la ejecución continúe en la primera instrucción que siga a la instrucción etiquetada.

4.10. El método *main()*

Toda aplicación Java está compuesta por al menos una clase, incluso la sencilla aplicación del *"Hola Mundo!"*. En al menos una clase, debe estar declarado con el modificador de acceso *public*, un método estático llamado *main()*, cuyo formato deber ser:

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

NOTA: A partir de la versión 1.5 de Java también lo podemos ver definido de la siguiente manera que explicaremos más adelante:

```
public static void main (String… args){}
```

El método *main()* debe cumplir las siguientes características:

- Ha de ser un método público, es decir, *public*.
- Ha de ser un método estático, es decir, *static*.
- No pude devolver ningún resultado, es decir, *void*.
- Ha de declarar un array de cadena de caracteres o un número variable de argumentos

El método *main()* es el punto de arranque de un programa Java, cuando se invoca al comando *java.exe* desde la línea de comandos, la JVM busca en la clase indicada un método estático llamado *main()*. Dentro del código de *main()* pueden crearse objetos de otros clases e invocar a sus métodos, en general, se puede incluir cualquier tipo de lógica que respete las restricciones indicadas para los método estáticos.

Es posible suministrar parámetros al método *main()* a través de la línea de comandos. Para ello, los valores a pasar deberán especificarse a continuación del nombre de la clase, separados por un espacio:

```
>java nombre_clase arg1 arg2 arg3
```

`class Argumentos{`
`  public static void main(String[] arg){`
`  System.out.println(arg[0]);`
`  System.out.println(arg[1]);`
`  System.out.println(arg[2]);`
`  System.out.println(arg[3]);`
 	
`  }`
`}`
	¿Cuál sería la salida?

```
java Ejemplo hola que tal
```

4.11. Arrays

Un *array* es un objeto en el que se puede almacenar un conjunto de datos de un mismo tipo. Cada uno de los elementos del *array* tiene asignado un índice numérico según su posición, siendo 0 el índice del primero.

4.11.1. Declaración

Un array debe declararse utilizando la expresión.

```
tipo [] variable_array;
ó
tipo variable_array[];
```

Como se puede apreciar, los corchetes pueden estar situados delante de la variable o detrás. Algunos ejemplos de declaración de *array* son:

```
int [] k;
String [] p;
char cads[];
```

Los *arrays* pueden declararse en los mismos lugares que las variables: como *atributos* de una clase o *locales* en el interior de un método. Como ocurre con cualquier otro tipo de variable objeto, cuando un *array* se declara como atributo se inicializa a *null;*

4.11.2. Dimensionado de un array

Para asignar un tamaño al array se utiliza la expresión:

```
variable_array = new tipo[tamaño];
```

También podemos hacer los pasos de declaración y dimensionado en una misma línea de declaración.

Cuando un *array* se dimensiona, todos sus elementos son inicializados explícitamente al valor por defecto del tipo correspondiente, independientemente de que la variable que contiene al array sea un atributo o sea local.

Existe una forma de declarar, dimensión e inicializar un array en una misma sentencia, sería:

```
int []nums = {1,2,3,4};
```

4.11.3. Acceso a los elementos de un array

El acceso a los elementos de un array se realiza utilizando la expresión.

```
variable_array[índice]
```

donde *índice* representa la posición a la que se quiere tener acceso, y cuyo valor deber estar comprendido entre 0 y *tamaño-1*.

Todos los objetos array tienen un atributo público, llamado *length*, que permite conocer el tamaño al que ha sido dimensionado un array. Este atributo resulta especialmente útil en aquellos métodos que necesitan recorrer todos los elementos de un array, independientemente de su tamaño.

Por ejemplo lo utilizaremos en una situación:

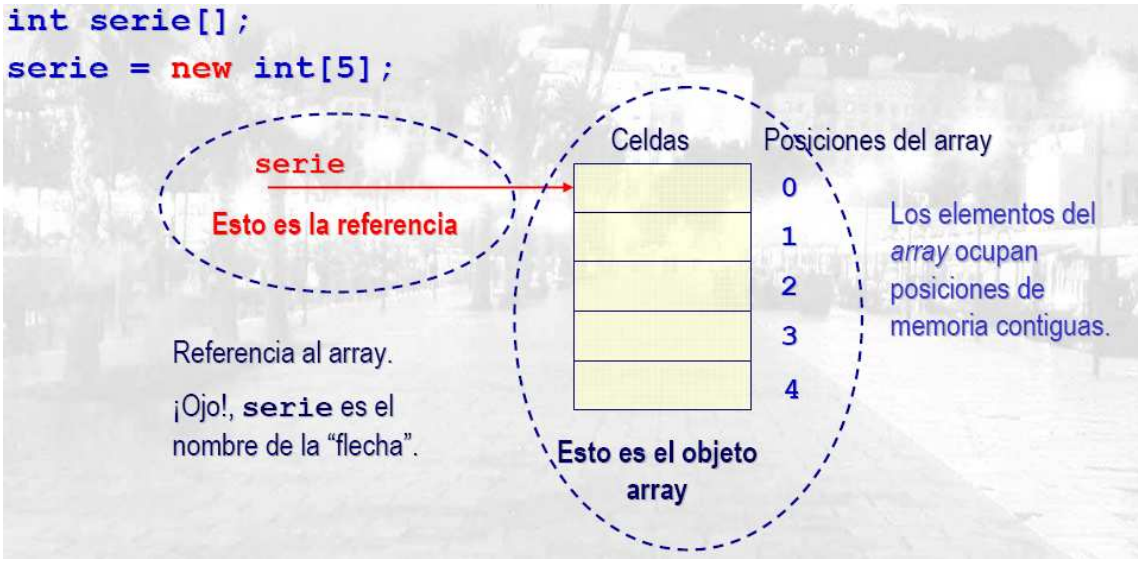
```
int []miArray = {1,2,3,4};
for(int i = 0;i<miArray.length;i++){
      System.out.println(miArray[i]);
}
```

```
class Arrays{
      public static void main(String... args){
            System.out.println(args.length);
            int []array;
            System.out.println(array.length);
      }
}

java Arrays
```

¿Cuál sería la salida?

Gráficamente:



4.11.4. Paso de un array como argumento de llamada a un método

Además de los tipos básicos los arrays también pueden utilizarse como argumentos de llamada a métodos. Para declarar un método que reciba como parámetro un array se emplea la sintaxis:

```
tipo método(tipo variable_array[]){
      //sentencias del método
}
```

En un ejemplo más práctico, el siguiente método recibe como parámetro un array de enteros:

```
public void metodoArrays(int []arr){
      System.out.println("…");
}
```

Para llamar a este método desde el *main()* o cualquier otra clase:

```
objeto.metodoArrays(variable_array);
```

La siguiente clase, calcula la suma de los elementos del array que le pasamos como argumento a un método de la propia clase:

```
class Arrays{

      public void sumaElementos(int []arr){
            int suma = 0; //muy importante inicializarla
            for(int i=0;i<arr.length;i++){
                  suma += arr[i];
            }
            System.out.println("SUMA: "+suma);
      }

      public static void main(String... args){
            int []miArray = {1,2,3,40};
            Arrays arr = new Arrays();
            arr.sumaElementos(miArray);
      }
}
```

```
C:\Documents and Settings\Administrador\Escritorio\raiz>javac Arrays.java

C:\Documents and Settings\Administrador\Escritorio\raiz>java Arrays
SUMA: 46
```

4.11.5. Recorrido de arrays con *for-each. Enhanced for Loop*

A partir de la versión Java 5, el lenguaje incorpora una variante de la instrucción *for*, que facilita el recorrido de arrays y colecciones (las veremos más adelante) para la recuperación de su contenido, eliminando la necesidad de utilizar una variable de control que sirva de índice para acceder a las distintas posiciones.

La sintaxis es la siguiente:

```
for(tipo variable: variable_array){
      //instrucciones
}
```

donde *tipo variable* representa la declaración de una variable auxiliar del mismo tipo que el array, de esta forma *variable* irá tomando cada uno de los valores almacenados en éste con cada iteración del *for*, siendo *variable_array* la variable que apunta al array.

```
public static void main(String... args){
      int []miArray = {1,2,3,4};
      for(int valor:miArray){
            miArray[valor] = 0;
      }

      for(int valor:miArray){
            System.out.println(valor);
      }
}
```

¿Cuál sería la salida?

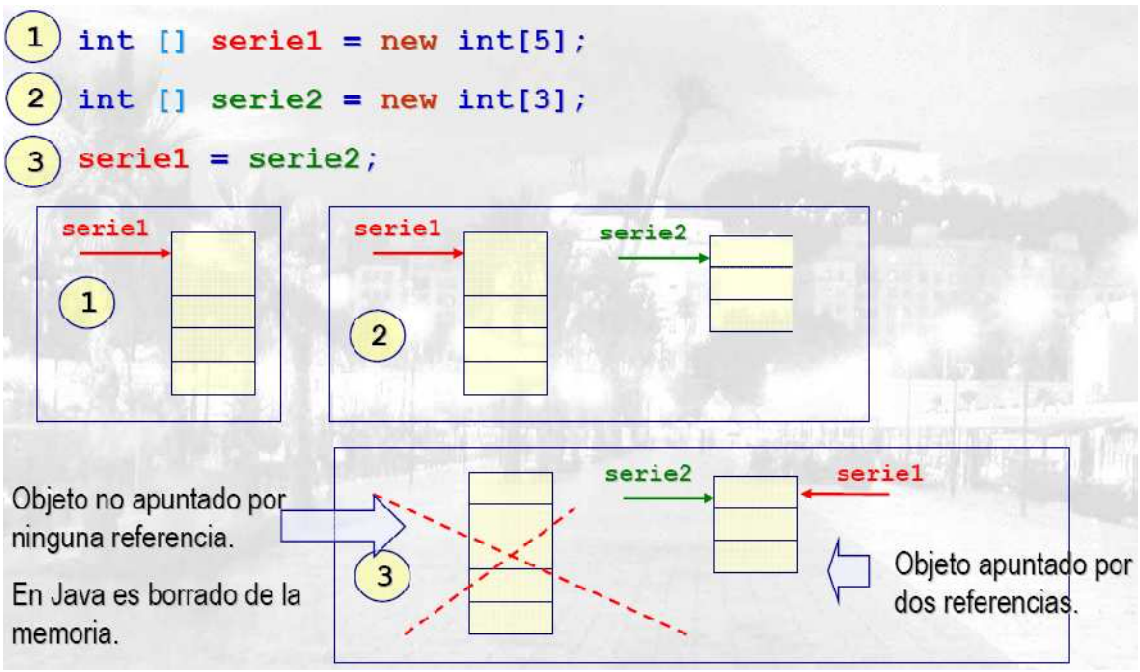
En la siguiente tabla podemos ver las diferencias sintácticas entre recorrer un array o colección con un *for* o un *for-each*:

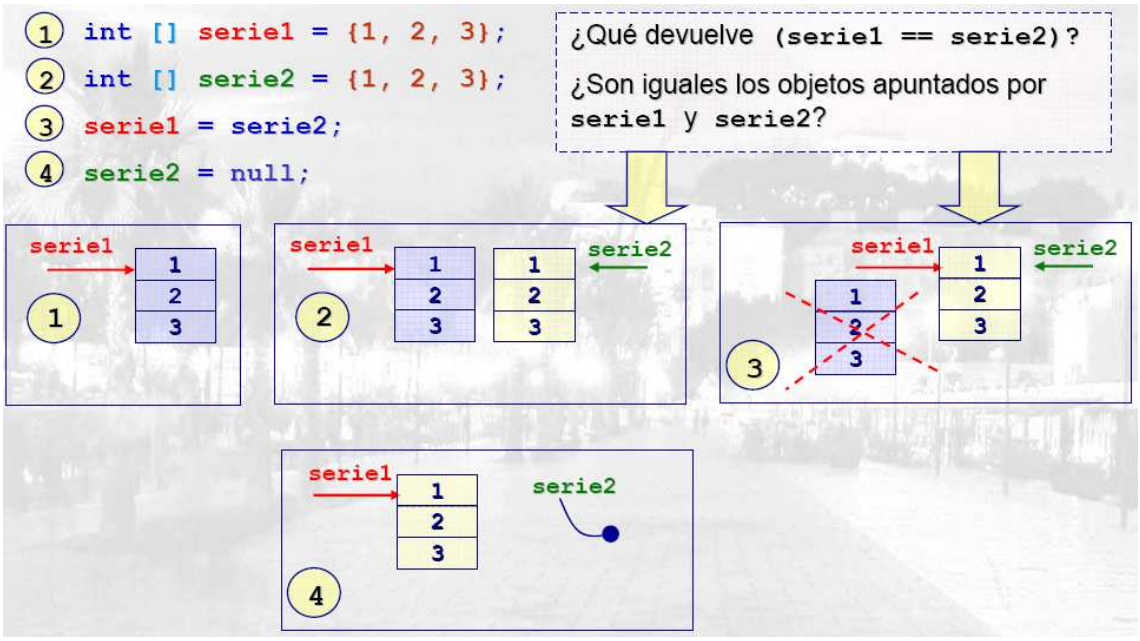
For-each loop	Equivalent for loop

`for (type var : arr) {`
`  body-of-loop`
`}`
	`for (int i = 0; i < arr.length; i++) {`
	`  type var = arr[i];`
	`  body-of-loop`
	`}`

`for (type var : coll) {`
`  body-of-loop`
`}`
	`for (Iterator<type> iter = coll.iterator();`
	`  iter.hasNext(); ) {`
	`  type var = iter.next();`
	`  body-of-loop`
	`}`

Algunas consideraciones extras sobre arrays:





4.11.6. Arrays multidimensionales

Los arrays en Java pueden tener más de una dimensión, por ejemplo, un array de enteros de dos dimensiones se declararía:

```
int [ ][ ] k;
```

A la hora de asignarle tamaño se procedería como en los de una dimensión indicando en los corchetes el tamaño de cada dimensión:

```
k = new int [3][5];
k[1][3] = 28;
```

Si imaginamos un array de dos dimensiones como una tabla organizada en filas y columnas, el array anterior tendría el aspecto siguiente:

k

	0	1	2	3	4
0					
1				28	
2					

4.11.7. Recorrido de un array multidimensionales

Para recorrer un array multidimensional podemos utilizar la instrucción *for* tradicional, empleando para ello tantas variables de control como dimensiones tenga el array. Por ejemplo:

```
int [][]miBi = new int[3][5];
miBi[1][3] = 28;
for(int i=0;i<3;i++){
	for(int j=0;j<5;j++){
		System.out.println("("+i+","+j+")");
	}
}

for(int i=0;i<miBi.length;i++){
	for(int j=0;j<miBi[i].length;j++){
		System.out.println("("+i+","+j+")");
	}
}

for(int n:k){
	System.out.println("valor"+n);
}
```

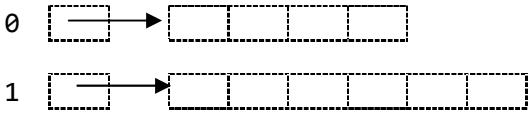
4.11.8. Arrays multidimensionales irregulares

Es posible definir un array multidimensional en el que el número de elementos de la segunda y sucesivas dimensiones sea variable, indicando únicamente el tamaño de la primera dimensión:

```
int [ ] [ ] p = new int [2][ ];
```

Un array de estas características equivale a un array de arrays, donde cada elemento de la primera dimensión almacenará su propio array:

```
p[0] = new int[4];
p[1] = new int [6];
```



4.11.9. Arrays de objetos

Tenemos que tener un poco de cuidado a la hora de crear un array de objetos en Java, ya que para estos se requiere una "doble instanciación", la primera de ellas sería para crear el espacio en memoria suficiente para albergar el array y la segunda se correspondería con la creación del objeto en memoria:

```
class ArrayObjetos{
	public int valor;
	public static void main(String... args){

		ArrayObjetos []ao = new ArrayObjetos[3];
		ao[0] = new ArrayObjetos();
		ao[0].valor = 0;
		ao[1] = new ArrayObjetos();
		ao[1].valor = 1;
		ao[2] = new ArrayObjetos();
		ao[2].valor = 2;
		for(ArrayObjetos aao:ao){
			System.out.println(aao.valor);
		}

		ArrayObjetos []ao2 = {new ArrayObjetos(),
			new ArrayObjetos(),new ArrayObjetos()};
		ao2[0].valor = 3;
		ao2[1].valor = 4;
		ao2[2].valor = 5;
		for(ArrayObjetos aao:ao2){
			System.out.println(aao.valor);
		}
	}
}
```

4.12. Tipos Enumerados

Los tipos enumerados son otra de las características incluidas en la versión 5 de J2SE, consistente en la posibilidad de definir nuevos tipos de datos cuyos posibles valores están limitados a un determinado conjunto dado.

Anteriormente a la versión 5, la forma de definir un determinado conjunto de valores era mediante la utilización de constantes, de la siguiente manera:

```
public class ClaseConstantes{
      public static final int YES=1;
      public static final int NO=0;
}
```

De esta forma, la manera de acceder a las mismas sería:

```
NombreClase.CONSTANTE;  ClaseConstantes.YES;
```

Mediante los tipos enumerados definimos un conjunto de posibles valores que pueden almacenar una variable de ese tipo, generando un error de compilación la asignación a la variable de un valor no definido en la enumeración.

4.12.1. Definición de un tipo enumerado

Un tipo enumerado se define según el siguiente formato:

```
[public] enum Nombre_tipo {VALOR1, VALOR2…}
```

siendo `Nombre_tipo` el nombre que se va asignar al tipo enumerado y `VALORN` los posibles valores que pueden tomar. La declaración del tipo enumerado puede estar en el interior de una clase o en el exterior, pero **nunca en el interior de un método**.

Por ejemplo:

```
enum ClaseConstantes{YES,NO}   //NO VA ; NO ES OBLIGATORIO
```

En nuestra clase declararíamos un atributo privado de tipo ClaseConstante, con lo cual el compilador solo admitirá que se le asigne los valores definidos en dicha enumeración, haciendo de esta forma que el código sea más seguro.

4.12.2. Clase Enumeración

La enumeración también es en sí una clase especie que hereda de *java.lang.Enum*. A diferencia de las clases estándares, una clase enumeración no permite el uso del operador *new* para la creación de objetos de la misma. Cada uno de los valores de la enumeración representa uno de los posibles objetos de la clase, así pues, éstos serán creados de forma implícita al hacer referencia en el código de estos valores.

Además de los métodos heredados por *Enum*, todas las enumeraciones disponen de un método estático *values()*, que devuelve un array con todos los objetos de la clase.

Otra diferencia entre los objetos de la clases estándares y los objetos enumerados está en que estos últimos pueden ser utilizados en expresiones de comparación de igualdad mediante el signo == (igual igual) y como valores de una instrucción *switch*.

Constructores y métodos de una enumeración

Por ser muy parecido a una clase, en esta se podrán definir constructores y métodos. Tanto unos como otros declarados después de la lista de valores de la enumeración.

CONSTRUCTORES Y METODOS

Se definen de la siguiente manera:

```
Nombre_enumeracion (lista_parametros){
      …
}
```

Cuando una enumeración dispone de constructores con parámetros, los valores de los argumentos de llamada **deben estar especificados en cada de los valores de la enumeración**:

```
enum MiEnumeracion{ //AUNQUE ES CLASE NO LLEVA class
      YES(1), NO(0)

      int valor;

      //Constructor
      MiEnumeracion(int v){
            valor = v;
      }

      //Metodo
      int getValor(){
            return valor;
      }
}
```

¿Cuál sería la salida?

```
1  class PruebaEnum2{
2        public Object[] obtenerObjetos(){
3              enum PuntosCardinales{
4                    NORTE,SUR,ESTE,OESTE
5              }
6              return PuntosCardinales.values();
7        }
8        public static void main(String... args){
9              PruebaEnum2 pe = new PruebaEnum2();
10
11             Object []obj = pe.obtenerObjetos()
12             for(Object o:obj){
13                   System.out.println(o.toString());
14             }
15       }
16 }
```

¿Cuál sería la salida?

```
1  enum ClaseConstantes{
2        YES(1),
3        NO(0);
4        private int valor;
5        ClaseConstantes(int valor){
6              this.valor = valor;
7        }
8        public int getValor(){
9              return valor;
10       }
11       public static void main(String []args){
12             System.out.println("HOLA MAIN");
13       }
14 }
```

```
1  class ProbarClaseCosntante{
2        ClaseConstantes cc;
3        public static void main(String... args){
4              ClaseConstantes c1 = new ClaseConstantes(1);
5              int valorYES = c1.YES.getValor();
6              ProbarClaseCosntante pcc = new ProbarClaseCosntante();
7              ClaseConstantes []valores = pcc.cc.values();
8              for(ClaseConstantes v:valores){
9                    System.out.println(v.toString()+"-"+v.getValor());
10             }
11       }
12 }
```

4.13. Métodos con número variable de argumentos

Si echamos un vistazo a los ejemplos vistos en los diferentes puntos hasta ahora, a la hora de llamar a un método de una clase necesitamos, primero si el método no es estático tenemos que crearnos una instancia de la clase (lo veremos en el siguiente punto) y a la hora de llamar al método, utilizaremos esa instancia seguida de un punto y el nombre del método pasándole un número determinado de parámetros, esto significa, que si el método tiene dos parámetros, la llamada al método deberá efectuarse con dos argumentos.

A partir de la versión 5 de Java es posible definir métodos que reciban un número variable de argumentos, para ello es necesario declarar un parámetro que recoja todos los argumentos de número variable. La declaración de un parámetro de estas características debe hacerse utilizando el siguiente formato:

```
tipo … nombre_parametro
```

Por ejemplo, un parámetro declarado de la siguiente forma en un método:

```
String … cadenas
```

Un array declarado de esta manera es realmente un array en el que se almacena los argumentos recibidos y cuyos valores deben de ser todos del mismo tipo.

```
class ArgumentosVariables{
	public int sumador(int... valores){
		//muy importante inicializar
		int suma = 0;
		for(int sum:valores){
			suma +=sum;
		}
		return suma;
	}

	public static void main(String... args){
		ArgumentosVariables av = new ArgumentosVariables();
		int suma1 = av.sumador(1,2);
		int suma2 = av.sumador(1,2,3,4);
		int suma3 = av.sumador(1,2,3,4,5,6);

		System.out.println("SUMA1: "+suma1);
		System.out.println("SUMA2: "+suma2);
		System.out.println("SUMA3: "+suma3);
	}
}
```

Salida:

```
C:\Documents and Settings\Administrador\Escritorio\raiz>java ArgumentosVariables
SUMA1: 3
SUMA2: 10
SUMA3: 21
```

Un método puede declarar tantos parámetros estándares como parámetros para número variable de argumentos. En estos casos, los segundos **deben de aparecer al final** de la lista de parámetros:

```
public void metodo1(int k, String s, int… nums){} //CORRECTO
public void metodo2(int p, String… cads, long f){} //INCORRECTO
```

La definición de métodos con un número variable de argumentos resulta muy útil para todas aquellas situaciones en donde la cantidad de datos que deba recibir el método en la llamada no esté determinada.

```
1  class ArgumentosVariables{
2
3        public int sumador(int... valores){
4              //muy importante inicializar
5              int suma = 0;
6              for(int sum:valores){
7                    suma +=sum;
8              }
9              return suma;
10       }
11       public int sumador(int a, int b){
12             int suma = 0;
13             suma = a + b + 20;
14             return suma;
15       }
16
17       public int sumador(short a, short b, short c,short d){
18             int suma = 0;
19             suma = a + b + c + d + 400;
20             return suma;
21       }
22
23       public static void main(String... args){
24             ArgumentosVariables av = new ArgumentosVariables();
25             int suma1 = av.sumador(1,2);
26             int suma2 = av.sumador(1,2,3,4);
27             int suma3 = av.sumador(1,2,3,4,5,6);
28
29             System.out.println("SUMA1: "+suma1);
30             System.out.println("SUMA2: "+suma2);
31             System.out.println("SUMA3: "+suma3);
32       }
33 }
```

5. Clases de uso general

De momento solo nos hemos centrado en la sintaxis del lenguaje. Se han visto desde los datos básicos y su manipulación mediante variables, operadores, etc. También hemos analizado las instrucciones de control que proporciona el lenguaje y hemos creado y manejado arrays.

En algún ejemplo hemos visto el uso de los objetos de las propias clases que hemos creado nosotros, pero todavía no hemos hecho un verdadero uso de todas las clases estándares que nos proporciona el lenguaje (más de 8000 entre clases e interfaces), con sus correspondientes métodos.

Llega el momento de analizar en profundidad algunas de estas clases, sobre todo aquellas que se les conoce como de "uso general" y que nos van a permitir resolver problemas que se puedan plantear en cualquier tipo de aplicación.

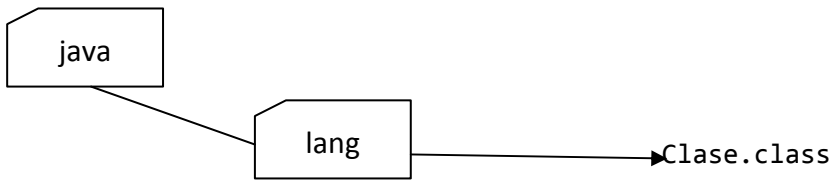
Vamos a dividirlas en:

- Clases básicas.
- Clases envoltorio (Wrappers).
- Clases de entrada/salida.
- Colecciones.

5.1. Organización de las clases: los paquetes

Los paquetes son un mecanismo utilizado por Java que permite organizar las clases de manera estructurada y jerárquica. Básicamente un paquete es una carpeta en la que se almacenan archivos .class; Además un paquete puede tener subpaquetes.

Todas las clases de Java siguen esta organización. En el caso de J2SE existe un paquete principal, llamado java (por eso nosotros no podemos crear un paquete llamado así), dentro del cual existen una serie de subpaquetes en donde se encuentran las distintas clases que componen J2SE.




```
java.lang.Clase
(paquete.subpaquete.clase)
```

5.1.1. Ventajas de utilizar paquetes

La utilización de paquetes para la organización de las clases en Java proporciona los siguientes beneficios:

- **Organizar las clases de manera estructurada**
  De la misma manera que almacenamos archivos en cualquier directorio de nuestro PC para mantener un orden, los paquetes permiten agrupar clases que tengan algún tipo de relación lógica, facilitando su localización y utilización en .un programa

- **Se evitan conflictos de nombres**
  Cuando tenemos una clase localizada en un paquete, la estamos identificando por un *nombre cualificado de este.* Como vimos antes se compone del nombre de la clase, precedido por los nombres de los subpaquetes en donde se encuentran hasta llegar al paquete principal, separado todo ello por "." (punto). Esto nos permite por ejemplo, que en nuestra aplicación pueden aparecer dos o más clases con el mismo nombre siempre que estén en diferentes paquetes.

5.1.2. Importar paquetes y clases

Cuando en una clase queremos hacer referencia a otra clase que se encuentra en otro paquete, es necesario utilizar el nombre cualificado de la misma. Esto puede parecer un poco engorroso, por ejemplo:

```
paquete.subpaquete1.subpaquete2.Clase obj =
      new paquete.subpaquete1.subpaquete2.Clase();
```

La solución a esto consiste en usar la importación de clases (siempre debe ser la primera línea de nuestro programa). Al importar la clase, podemos hacer referencia a la misma sin necesidad de utilizar el nombre cualificado, tan solo el nombre de la clase.

Para importar una clase se utiliza la sentencia import, de tal forma:

```
import paquete.subpaquete1.subpaquete2.Clase;
```

Si por el contrario lo que queremos hacer es importar varias clases que están dentro de ese paquete, haremos:

```
import paquete.subpaquete1.subpaquete2.*;
```

5.1.3. Paquetes de uso general

Todas las clases que vamos a ver en este tema se encuentran en los siguientes paquetes:

- **java.lang**
  Incluye las clases fundamentales para el desarrollo de cualquier aplicación en Java. Dado que sus clases son de uso común en los programas, *el compilador la importa por completo de forma implícita*, esto significa, que no tendremos que hacer un *import* de este paquete para hacer uso de las clases que contiene.

- **java.io**
  Contiene las clases para la gestión de la entrada/salida de datos en Java. Independientemente del dispositivo de E/S que se utilice, Java utiliza siempre las mismas clases para evitar datos a la salida y leer datos de la entrada.

- **java.util**
  En este paquete encontramos clases para utilidades varias, tales como el tratamiento de colecciones de objetos, la manipulación de fechas, la construcción de expresiones regulares.

5.2. Gestión de cadenas: String, StringBuffer, StringBuilder

En Java las cadenas de caracteres no se corresponden con ningún tipo básico, sino que son objetos pertenecientes a la clase *java.lang.String*.

La clase String proporciona una amplia variedad de métodos que permiten realizar las operaciones de manipulación y tratamiento de cadena de caracteres habituales en un programa.

5.2.1. La clase String

Un concepto a tener en cuenta con la clase String es que, *una vez creado un objeto de este tipo, no puede ser modificado.* El objeto es inmutable, pero la referencia que se crea no lo es.

Para crear un objeto de tipo String podemos seguir el procedimiento general de creación de objetos en Java, utilizando el operador new, por ejemplo:

```
String cad1 = new String("Juan Carlos");
```

Sin embargo, y dada la amplia utilización de estos objetos dentro de un programa, Java permite crear y asignar un objeto String a una variable de la misma forma que se hace con cualquier tipo de dato básico, por lo tanto, también podemos hacer:

```
String cad2 = "Juan Carlos";
```

Obviamente, además de la diferencia sintáctica hay implícita una diferencia en el tratamiento por parte de la JVM.

```
String cad1 = "Juan Carlos";
//Estamos creando un solo objeto (non-pool) y una referencia.

String cad2 = new String("Juan Carlos");
//Estamos creando dos objetos (in pool) y una referencia.
```

TRATAMIENTO DE LA MEMORIA

Para que el manejo de memoria sea más eficiente, la JVM reserva una porción de memoria conocida como "String Pool".

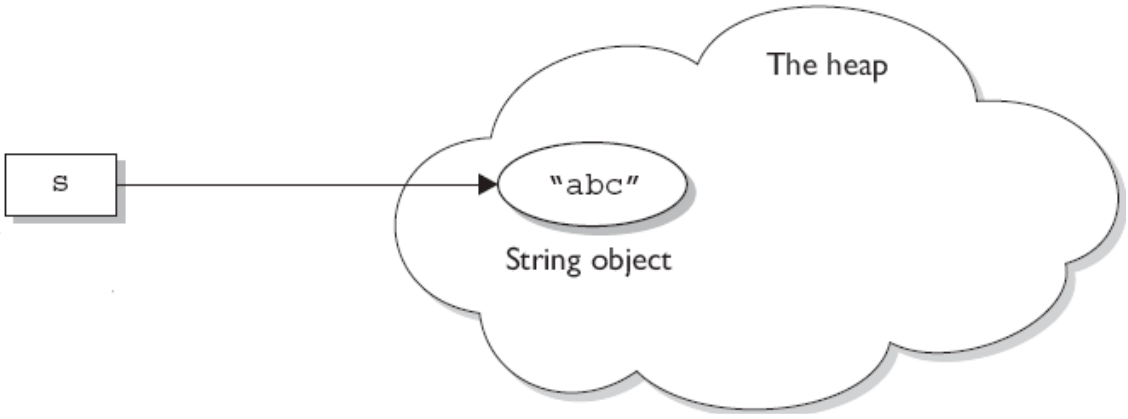
La regla es la siguiente: cuando el compilador encuentra un literal, se fija en el pool a ver si hay alguno que coincida, si es así la referencia es dirigida al literal existente, y no se crea un nuevo literal.

Esto es lo que hace a la inmutabilidad una buena idea:

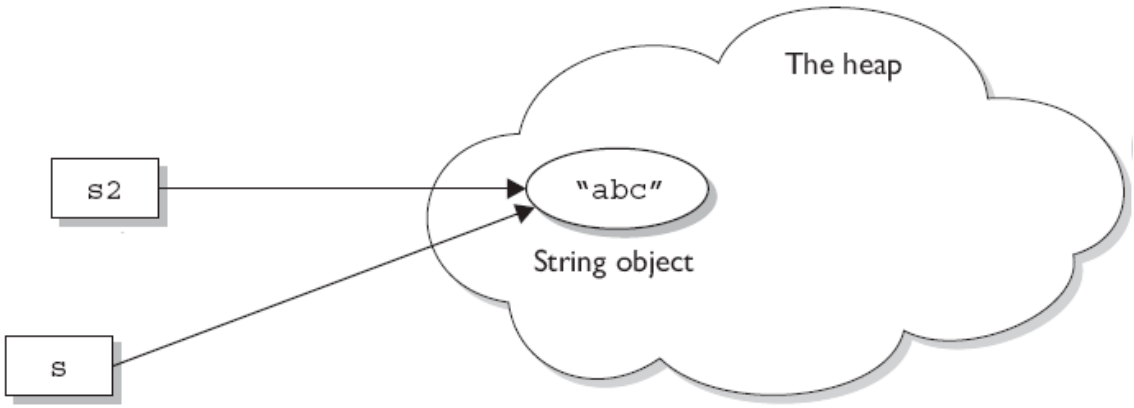
- Si una de varias referencias al mismo String pudiese modificarlo, sería desastroso.
- Pero… ¿Qué pasa si alguien sobrescribe la funcionalidad de la clase String? ¿Podría causar problemas en el pool? NO se puede porque la clase String está marcada como FINAL.

Vamos a explicar un poco más la inmutabilidad de la clase String a través de una serie de ejemplos. Supongamos que tenemos el siguiente código:

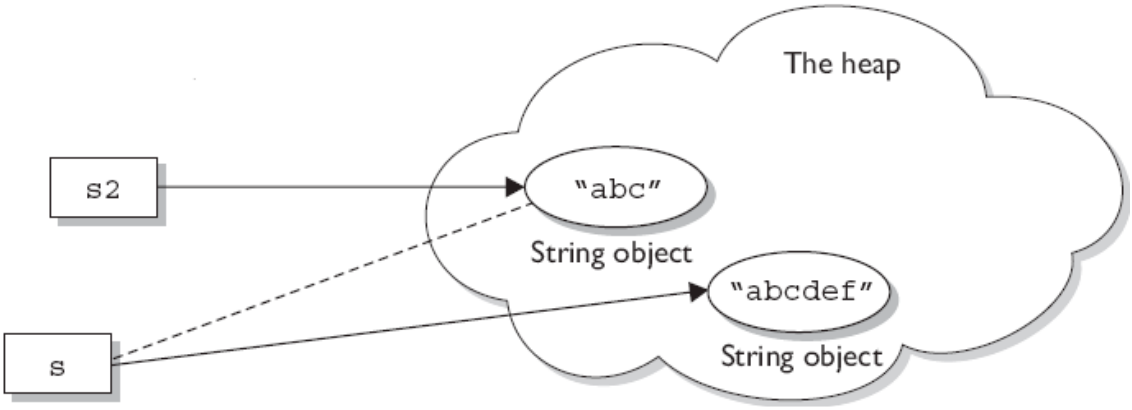
```
String s = “abc”;
```




```
String s2 = s;
```



```
s = s.concat("def");
```



Lo que realmente ocurre es que al concatenar "abc" con "def" se **está creando un nuevo objeto** "abcdef", que pasa a ser referenciado por la variable "s". En este caso si "s2" no siguiera apuntado a "abc" perderíamos toda posibilidad de volver a acceder a ese valor y actuaria el recolector de basura para liberar ese espacio de memoria ocupada.

Principales métodos de la clase String

La clase String cuenta con un amplio conjunto de métodos para la manipulación de cadenas de texto; a continuación se indican algunos de los más interesantes:

METODO	RETORNO	DESCRIPCION
charAt(int index)	char	Retorna el carácter ubicado en la posición index
concat(String s)	String	Retorna el String que resulta de concatenar el String utilizado para invocar el método y s.
equals(String) **equalsIgnoreCase(String s)**	boolean	Retorna true si el contenido de ambos Strings es igual, ignorando mayúsculas/ minúsculas
length()	int	La longitud del String usado para invocar el método
replace(char old,char new)	String	Retorna un String resultado de reemplazar el carácter "old" por el carácter "new"
substring(int a) **substring (int a,int b)**	String	Retorna una subcadena que va desde la posición 'a' hasta la posición 'b'.
toLowerCase()	String	Retorna un String cuyo valor es el del utilizado para invocar el método, pero con todas las mayúsculas intercambiadas por minúsculas .
toString()	String	El valor del String.
toUpperCase()	String	Funciona igual que toLowerCase, pero intercambiando minúsculas por mayúsculas.
trim()	String	Retorna un String cuyo valor es el del utilizado para invocar el método pero sin espacios al principio o al final de la cadena.

```
class PruebaCadenas1{
  public static void main(String... args){
    String cad1 = "Carlos";
    String cad2 = "Carlos";
    String cad3 = new String("Carlos");
    String cad4 = new String("Carlos");

    System.out.println(cad1 == cad2);
    System.out.println(cad1 == cad3);
    System.out.println(cad3 == cad4);
    System.out.println(cad1.equals(cad2));
    System.out.println(cad1.equals(cad3));
    System.out.println(cad3.equals(cad4));

    String x = new String("Juan Carlos");

    System.out.println(x.charAt(2));
    System.out.println(x.concat(" Gonzalez"));
    System.out.println(x.length());
    System.out.println(x.replace('z','Z'));
    System.out.println(x.substring(0,10));
    System.out.println(x.toLowerCase());
    System.out.println(x.trim());

  }
}
```

¿Cuál sería la salida?

OTROS METODOS

- `int` **`indexOf`**`(String cad)`
  Devuelve la posición de la cadena indicada en el parámetro, dentro de la cadena principal. Si no aparece, devuelve -1.

- `static String` **`valueOf`**`(tipo_basico dato)`
  Método estático que devuelve como cadena el valor pasado como parámetro. Existen tantas versiones de este método como tipos básicos en Java.

- `String[]` **`split`**`(String regex)`
  Devuelve un array de String resultante de descomponer la cadena de texto en subcadenas, utilizando como separador de elemento el carácter especificado en el parámetro por regex.

```
class PruebaCadenas1{
  public static void main(String... args){
    String cad1 = "Carlos";
    String cad2 = "Carlos";
    String cad3 = new String("Carlos");
    String cad4 = new String("Carlos");

    System.out.println(cad1 == cad2);
    System.out.println(cad1 == cad3);
    System.out.println(cad3 == cad4);
    System.out.println(cad1.equals(cad2));
    System.out.println(cad1.equals(cad3));
    System.out.println(cad3.equals(cad4));

    String x = new String("Juan Carlos");

    System.out.println(x.charAt(2));
    System.out.println(x.concat(" Gonzalez"));
    System.out.println(x.length());
    System.out.println(x.replace('z','Z'));
    System.out.println(x.substring(0,10));
    System.out.println(x.toLowerCase());
    System.out.println(x.trim());

  }
}
```

¿Cuál sería la salida?

5.2.2. La clase StringBuffer

La clase StringBuffer aparece para hacer frente a la inmutabilidad de los objetos String, haciendo posible la modificación de los mismos. Un objeto StringBuffer representa una cadena de texto modificable.

Para crear un objeto de esta clase hay que hacerlo explícitamente a través del operador *new*, **no admitiendo la asignación directa** como ocurría en el caso de String, por lo tanto:

```
StringBuffer bf = new StringBuffer("Cadena Buffer");
```

Cuidado porque StringBuffer, **no admite el operador "+"** para la concatenación de cadenas:

```
StringBuffer bf = new StringBuffer("Cadena");
StringBuffer bf2 = bf + "mas cadena";
```

Salida

La clase StringBuffer se utiliza habitualmente en operaciones de lectura de grandes cantidades de datos desde algún dispositivo externo, ya que resulta más eficiente ir anexando los datos leídos en único objeto que crear objetos de texto parciales en cada lectura.

Los métodos más importantes de esta clase son:

- `StringBuffer` **`append`** `(String cad)`
  Añade al objeto StringBuffer la cadena suministrada como parámetro. Hay otras versiones con diferente argumento: boolean, int, float. Devuelve una referencia al propio objeto StringBuffer.

- `StringBuffer` **`insert`** `(int pos, String cad)`
  Inserta dentro del objeto original, en la posición especificado por *pos*, la cadena *cad*.

- `StringBuffer reverse ()`
  Invierte la cadena StringBuffer, devolviendo una nueva referencia al objeto.

Una cosa importante que tenemos que tener en cuenta sobre la clase *StringBuffer* es que no sobrescribe el método `equals()`, por lo tanto, si aplicamos este método para comparar dos referencias de *StringBuffer*, el resultado será true solamente si ambas referencias apuntan al mismo objeto.

```
StringBuffer bf1 = new StringBuffer("Cadena");
StringBuffer bf2 = new StringBuffer("Cadena");
if(bf1.equals(bf2)){
  System.out.println("iguales");
}
else{
  System.out.println("diferentes");
}
```

Salida

5.2.3. La clase StringBuilder

La clase *StringBuilder* dispone de los mismos métodos que *StringBuffer* y por lo tanto, tiene la misma utilidad.

La principal diferencia entre ambas esta en el hecho de que mientras los métodos de StringBuffer son *syncronized* para poder ser utilizados en aplicaciones multihilo, los de StringBuilder no lo son, mejorando el rendimiento de las aplicaciones de un único hilo.

5.3. La clase Arrays

En *java.util* se encuentra la clase *Arrays*, capaz de mantener un conjunto de métodos estáticos que llevan a cabo funciones de utilidad para arrays. Tiene cuatro funciones básicas:

- `equals( )` para comparar la igualdad de dos arrays;
- `fill( )` para rellenar un array con un valor;
- `sort( )` para ordenar el array;
- `binarySearch( )` para encontrar un dato en un array ordenado.

Todos estos métodos están sobrecargados para todos los tipos de datos primitivos y objetos. Además, hay un método simple `asList( )` que hace que un array se convierta en un contendor *List*, del cual se aprenderá más adelante.

5.3.1. Copiar un array

La biblioteca estándar de Java proporciona un método estático, llamado `System.arraycopy( )`, que puede hacer copias mucho más rápidas de arrays que si se usa un bucle *for* para hacer la copia a mano. `System.arraycopy( )` está sobrecargado para manejar todos los tipos.

```
public static void arraycopy(Object origen, int posOrig,
      Object dest, int posDest, int length)
```

- `origen:` Nombre del array origen del que queremos copiar elementos.
- `posOrig:` Posición del array origen desde la que queremos empezar a copiar.
- `dest:` Nombre del array destino (tiene que existir el array) al que queremos copiar elementos del array origen.
- `posDest:` Posición del array destino desde la que queremos empezar a copiar los elementos del array origen.
- `lenght:` Numero de elementos del array origen que queremos copiar en el array destino.

```
class CopiaArrays{
      public static void main(String... args){
            int []arrOri={1,2,3,4,5};
            int []arrDes={10,9,8,7,6};

            System.arraycopy(arrOri, 0, arrDes, 5, arrOri.length);

            for(int lis:arrDes){
                  System.out.printf("%d",arrDes);
            }
      }
}
```

5.3.2. Comparar arrays

La clase Arrays proporciona un método sobrecargado `equals( )` para comparar arrays y ver si son iguales. Otra vez, se trata de un método sobrecargado para todos los tipos de datos primitivos y para Objetos.

Para que dos arrays sean iguales:

- deben tener el mismo número de elementos.
- cada elemento debe ser equivalente a su elemento correspondiente en el otro array, utilizando el método `equals( )` para cada elemento. (En el caso de datos primitivos, se usa la clase de su envoltorio `equals( );` por ejemplo, se usa `Integer.equals( )` para int.) .

Sintaxis:

```
public static boolean equals(tipo[] a, tipo[] a2)
```

Ejemplo:

```
import java.util.Arrays;
	class CompararArrays{

	private static int []arr1;
	private static int []arr2;

	public static boolean igualesObj(Integer []obj1, Integer []obj2){
		boolean valor = false;
		valor = Arrays.equals(obj1,obj2);
		return valor;
	}

	public static void main(String... args){
		int []arrOri={1,2,3,4,5};
		int []arrDes={1,2,3,4,5};

		boolean iguales1 = false, iguales2 = false, iguales3 = false;
		iguales1 = Arrays.equals(arrOri,arrDes);
		System.out.println("IGUALES:"+iguales1);

		iguales2 = Arrays.equals(arr1,arr2);
		System.out.println("IGUALES:"+iguales2);

		Integer []a1 = new Integer[]{1,2,3};
		Integer []a2 = new Integer[]{3,2,1};

		iguales3 = CompararArrays.igualesObj(a1,a2);
		System.out.println("IGUALES:"+iguales3);
	}
}
```

5.3.3. Comparaciones de elementos de arrays

En Java 2 hay dos formas de proporcionar funcionalidad de comparación. La primera es con el método de comparación natural, que se comunica con una clase implementando la interfaz `java.lang.Comparable`. Se trata de una interfaz bastante simple con un único método `compareTo( )`. ***Para que un objeto sea comparable, su clase debe implementar la interfaz `java.lang.Comparable`.***

Este método toma otro Objeto como parámetro, y produce un *valor negativo si el parámetro es menor que el objeto actual, cero si el parámetro es igual, y un valor positivo si el parámetro es mayor que el objeto actual*.

En el siguiente ejemplo podemos ver como comparamos los elementos del array entre si y finalmente mostramos el array ordenado:

```
import java.util.Arrays;

class CompararElementos implements Comparable{

	private static int []arrDes={10,9,8,7,6,2,1,3,4,5};

	public int compareTo (Object rv) {
		return ((CompararElementos) rv).compareTo(rv);
	}

	public static void main(String... args){
		Arrays.sort(arrDes);
		System.out.println(Arrays.toString(arrDes));
	}
}
```

Si no queremos utilizar el método de comparación natural tendremos que implementar la interface `java.lang.Comparator`, que posee un solo método `compare()`. Este método recoge los dos objetos que van a compararse como argumentos y devuelve un entero negativo si el primer argumento es menor que el segundo, cero si son iguales y un entero positivo si el primer argumento es más grande que el segundo

Veamos un ejemplo:

```
import java.util.Arrays;
import java.util.Comparator;

class CompararElementos2 implements Comparator{

	private static String []arrDes={"Juan","Carlos"};
	private static String []arrOri={"Carlos","Juan"};

	public int compare (Object obj1, Object obj2) {

		String cad1 = ((String)obj1).toLowerCase();
		String cad2 = ((String)obj2).toLowerCase();

		return cad1.compareTo(cad2);
	}

	public static void main(String... args){
		//Necesitamos un objeto de esta clase
		CompararElementos2 ce2 = new CompararElementos2();
		Arrays.sort(arrDes,ce2);

		//Array ordenador
		System.out.println(Arrays.toString(arrDes));

		//Tenemos que utilizar el mismo objeto
		int loc = Arrays.binarySearch(arrDes, arrOri[0],ce2);
		System.out.println("Posicion de: "+arrOri[0]+" en "+loc);
	}
}
```

Haciendo el moldeo a String, el método *compare()* implícitamente se asegura que solamente se están utilizando objetos de tipo String. Luego se fuerzan los dos Strings a minúsculas y el método *String.compareTo()* devuelve el resultado que se desea.

A la hora de utilizar un *Comparator* propio para llamar a *sort()*, se debe utilizar el mismo *Comparator* cuando se vaya a llamar a *binarySearch()*. La clase *Arrays* tiene otro método *sort()* que toma un solo argumento: un array de *Object*, sin ningún Comparator. Este método también puede comparar dos *Object*, utilizando el *método natural de comparación* que es comunicado a la clase a través del interfaz *Comparable*. Este interfaz tiene un único método, *compareTo()*, que compara el objeto con su argumento y devuelve negativo, cero o positivo dependiendo de que el objeto sea menor, igual o mayor que el argumento.

5.3.4. Ordenar un array

Con los métodos de ordenación incluidos, se puede ordenar cualquier array de tipos primitivos, y un array de objetos que, o bien implemente *Comparable*, o bien tenga un *Comparator* asociado. Éste rellena un gran agujero en las bibliotecas Java -se crea o no, en Java 1.0 y 1.1 no había soporte para ordenar cadenas de caracteres!.

Vamos con un ejemplo:

```
import java.util.Arrays;
class OrdenarArray{

	public static void main(String... args){

		String []arr1 = new String[]{"Juan","carlos","Zarza","Gonzalez"};

		System.out.println("ANTES DE ORDENAR");
		System.out.println(Arrays.toString(arr1));

		Arrays.sort(arr1);
		System.out.println("DESPUES DE ORDENAR");
		System.out.println(Arrays.toString(arr1));
	}
}
```

Salida

Algo que uno notará de la salida del algoritmo de ordenación de cadenas de caracteres es que es ***lexicográfico,*** por lo que coloca en primer lugar las palabras que empiezan con letras mayúsculas, seguidas de todas las palabras que empiezan con minúsculas. (Las guías telefónicas suelen ordenarse así.) También se podría desear agrupar todas las palabras juntas independientemente de si empiezan con mayúsculas o minúsculas, lo cual se puede hacer definiendo una clase *Comparator*, y por consiguiente, sobrecargando el comportamiento por defecto de Comparable para cadenas de caracteres.

```
import java.util.Arrays;
class OrdenarArray{

	public static void main(String... args){
		boolean []arrB = new boolean[]{true, false, true};
		Arrays.sort(arrB);
		System.out.println(Arrays.toString(arrB));
	}
}
```

```
Salida:
```

5.4. Utilización de fechas

5.4.1. La clase Date

La clase *Date* nos proporciona una fecha y una hora concreta con una precisión de milisegundos. A través de esta clase vamos a manipular la fecha y obtener información de la misma de diferentes formas. A partir de la versión Java 1.1 se incorporo una nueva clase *Calendar* que amplía las posibilidades a la hora de manejar la fecha.

El constructor por defecto *Date* es:

```
Date fecha = new Date();
```

A través del método *toString()* obtendríamos una representación de forma cadena de la fecha y hora actuales. Para la siguiente clase:

```
import java.util.Date;
class Fecha{

      public static void main(String... args){
            Date fecha = new Date();
            System.out.println(fecha.toString());
      }
}
```

```
Salida:

Thu Jul 23 12:08:41 CEST 2009
```

Internamente, el objeto *Date* guarda la información de la fecha y la hora como un numero de tipo *long* que representa la cantidad de milisegundos transcurridos desde el dia 1 de enero de 1970 hasta el momento de la creación del objeto. Numero que podemos obtener a través del método `getTime()` de la clase Date.

```
import java.util.Date;
class Fecha{

      public static void main(String... args){
            Date fecha = new Date();

            System.out.println(fecha.getTime());
            System.out.println((fecha.getTime()/(1000*60*60*24))/365);
      }
}
```

```
Salida:

Thu Jul 23 12:43:07 CEST 2009
1248345787906
39
```

5.4.2. La clase Calendar

Clase que nace como complemento a la clase *Date* para cubrir las carencias de esta. Para crear un objeto de este tipo, tenemos que tener en cuenta que se trata de una clase abstracta (hablaremos de ellas más adelante), y por lo tanto, no podemos instanciarla para crear un objeto, sino que lo tenemos que crear a partir de un método estático que posee `getInstance()`.

```
Calendar cal = Calendar.getInstance();
```

Ahora con el método `get()` podríamos recuperar de manera individual cada uno de los campos que componen la fecha. Éste método acepta un numero entero que indica el campo en concreto que queremos recuperar. La propia clase define una seria de constantes con los valores que corresponden a cada uno de los campos que componen la fecha.

Por ejemplo:

```
import java.util.Calendar;

class Calendario{

	public static void main(String... args){
		StringBuilder fecha = new StringBuilder();
		Calendar cal = Calendar.getInstance();
		fecha.append(cal.get(Calendar.DAY_OF_MONTH));
		fecha.append("/");
		fecha.append(cal.get(Calendar.MONTH)+1);
		fecha.append("/");
		fecha.append(cal.get(Calendar.YEAR));

		System.out.println(fecha);
	}
}
```

```
Salida:

23/7/2009
```

Como curiosidad podemos ver que en el caso del mes el valor devuelto por `get()` está comprendido entre 0 y 11, por lo que tendremos que sumarle 1 para obtener correctamente el mes en el que nos encontramos.

Métodos de la clase Calendar

Además del método `get()` que hemos visto antes, la clase *Calendar* nos proporciona los siguientes métodos para el manejo de fechas:

- `void set(int año, int mes, int dia)`: Modifica la fecha del objeto *Calendar*, asignándole el mes, año y día especificados en los parámetros. Este método está sobrecargado, existiendo otro método en el que le podemos dar valores tanto a la fecha como a la hora.
- `void setTime(Date d)`: Establece la fecha y hora del objeto *Calendar* a partir de un objeto *Date*.
- `Date getTime ()`: Devuelve la fecha/hora como un objeto *Date*.
- `void add (int campo, int cantidad)`: Realiza una modificación relativa en uno de los campos de la fecha/hora, añadiendo una cantidad de tiempo al campo especificado en el primer parámetro.
- `void roll (int campo, int cantidad)`: El funcionamiento es el mismo que el de *add()*, con la diferencia de que la adicción de la cantidad de tiempo no afectara a todos los campo, sino solo al campo sobre el que se aplica.
- `boolean equals(Object obj)` : Compara dos fechas, devuelve true si son exactamente iguales (a nivel de milisegundo, no de día), false de lo contrario.
- `boolean before(Date when)` : Devuelve true si una fecha es anterior a otra.
- `boolean after(Date when)` : Devuelve true si una fecha es posterior a otra.

Algunos valores estáticos, para *Calendar* son:

- **YEAR**: Año.
- **MONTH**: Mes. (Empiezan en 0)
- **DATE, DAY_OF_MONTH**: Día del mes.
- **DAY_OF_WEEK**: Día de la semana entre 1 (MONDAY) y 7 (SATURDAY).
- **HOUR**: Hora antes o después del medio día (en intervalos de 12 horas).
- **HOUR_OF_DAY**: Lo hora absoluta del día (en intervalos de 24 horas).
- **MINUTE**: El minuto dentro de la hora.
- **SECOND**: El segundo dentro del minuto.

```
import java.io.*;
import java.util.*;

class MetodosCalendar{

	public StringBuffer cadena (Calendar cal){
		StringBuffer fechaCompleta = new StringBuffer();
		fechaCompleta.append(cal.get(Calendar.YEAR));
		fechaCompleta.append("/");
		fechaCompleta.append(cal.get(Calendar.MONTH));
		fechaCompleta.append("/");
		fechaCompleta.append(cal.get(Calendar.DATE));

		System.out.printf("GC: %s\n",fechaCompleta);
		return fechaCompleta;
	}

	public static void main(String... args){
		MetodosCalendar mc = new MetodosCalendar();
		Calendar cal1 = Calendar.getInstance();

		cal1.set(2009,7,31);
		mc.cadena(cal1);

		Date d = new Date();
		Calendar cal2 = new GregorianCalendar();

		cal2.set(cal2.get(Calendar.YEAR),(cal2.get(Calendar.MONTH)+1)
				,cal2.get(Calendar.DATE));
		mc.cadena(cal2);

		/*Suponiendo:
				cal2: 2009/7/31
				cal1: 2009/7/30*/

		System.out.println(cal2.before(cal1));
		System.out.println(cal2.after(cal1));
		System.out.println(cal2.equals(cal1));

		cal1.add(Calendar.MONTH, 6);
		System.out.println(cal1.getTime().toString());

		cal2.roll(Calendar.MONTH, 6);
		System.out.println(cal2.getTime().toString());

	}
}
```

Salida:

5.4.3. La clase DateFormat

Para esta clase vamos a tener en cuenta:

- La clase `java.text.DateFormat` se utiliza para obtener una determinada fecha en forma de cadena de texto, utilizando un formato personalizado.
- Se trata de una clase abstracta, una de sus implementaciones es `java.text.SimpleDateFormat.`
- El método estático `getInstance()` nos permite obtener un objeto `DateFormat` con los datos por defecto de la máquina en la que se ejecuta el programa (idioma, localización, etc.).
- Las distintas variantes del método `format` nos permiten obtener la fecha/hora.

Formateo y análisis de fechas para la localización por defecto

En el caso de utilizar la localización o zona geográfica predeterminada, emplearemos:

```
DateFormat getDateInstance(int style);
```

donde el parámetro *style* representa el formato de fecha utilizado. Los posibles valores que pueden pasarse como argumentos de llamada a este método corresponden a las siguientes constantes definidas para *DateFormat*:

- SHORT
- MEDIUM
- LONG
- FULL
- DEFAULT ~MEDIUM

Así la siguiente instrucción crearía un objeto *DateFormat* para la localización predeterminada, con un tipo de formato para la fecha:

```
DateFormat df = DateFormat.getDateInstance(DateForma.LONG);
```

Una vez creado el objeto, se puede hacer uso del método **`format()`** para aplicar sobre una determinada fecha el formato definido por el objeto. La sintaxis sería la siguiente:

```
String format(Date fecha)
```

Ejemplo:

```
import java.text.DateFormat;
import java.util.Date;

class FormateoFechas{

	public static void main(String... args){

		DateFormat []df = new DateFormat[5];
		df[0] = DateFormat.getDateInstance(DateFormat.SHORT);
		df[1] = DateFormat.getDateInstance(DateFormat.MEDIUM);
		df[2] = DateFormat.getDateInstance(DateFormat.LONG);
		df[3] = DateFormat.getDateInstance(DateFormat.FULL);
		df[4] = DateFormat.getDateInstance(DateFormat.DEFAULT);

		for(DateFormat mdf: df){
			Date fecha = new Date();
			System.out.printf("%s\n",mdf.format(fecha));
		}
	}
}
```

Salida:

```
(>java FormateoFechas
3/08/09
03-ago-2009
3 de agosto de 2009
lunes 3 de agosto de 2009
03-ago-2009
```

Por el contrario, si lo que queremos es analizar una fecha, tendremos que utilizar el método **`parse()`** de *DateFormat*, cuya sintaxis es:

```
Date paser(String fecha)
```

Este método recibe como parámetro una cadena de caracteres que representa la fecha que se quiere analizar, devolviendo el objeto *Date* correspondiente a dicha fecha. Si la cadena de caracteres no tiene un formato valido, la llamada a *parse()* lanzara una excepción de tipo *ParseException*.

Para el uso de *parse()* tenemos que tener en cuenta la siguiente regla, y es que la cadena de caracteres pasada como argumento al método debe tener el formato especificado para el objeto DateFormat:

```
System.out.printf("%s\n",df[2].format(df[2].parse("3/08/2009")));
```

Formateo y análisis de fechas para la localización para una localización especifica

Para obtener un objeto *DateFormat* para una localización geográfica específica, deberemos emplear la siguiente versión sobrecargada del método *getDateInstance():*

```
DateFormat getDateInstance(int style, Locale area);
```

El parámetro *style* sigue teniendo el mismo significado que antes, mientras que *area* representa un objeto de la clase `java.util.Locale` asociado a una determinada cultura, zona o localización geográfica.

Existen varios objetos *Locale* predefinidos correspondientes a determinadas localizaciones geográficas que se incluyen como constantes en la clase *Locale*:

- UK: Reino Unido.
- GERMANY: Alemania.

Pero también podemos crear objetos de tipo *Locale* asociado a una cultura específica, para ello utilizaremos alguno de los siguientes constructores:

```
Locale (String language)
Locale(String language, String country)
```

Donde *language* es una cadena de texto asociada a una determinada lengua y *country* es el código del país. En el siguiente enlace podemos encontrar los códigos de lengua y país respectivamente:

http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm [Última vez visitado: 03/08/2009]

```
import java.text.DateFormat;
import java.util.*;

class Internacional{
	public static void main(String... args){
		DateFormat []df = new DateFormat[2];
		Locale []loc = new Locale[2];

		loc[0] = new Locale("it");
		loc[1] = new Locale("nl");

		df[0] = DateFormat.getDateInstance(DateFormat.FULL,loc[0]);
		df[1] = DateFormat.getDateInstance(DateFormat.FULL,loc[1]);

		for(DateFormat mdf: df){
			Date fecha = new Date();
			System.out.printf("%s\n", mdf.format(fecha));
		}
	}
}
```

SimpleDateFormat

Al mostrar una fecha en lugar de utilizar, `Mon Aug 03 14:16:13 CEST 2009`, podemos usar un formato más entendible, para este propósito se utiliza la clase *SimpleDateFormat*. Este objeto está diseñado para dar formato a fechas, para ello nosotros le ***especificamos un patrón*** de formato para una fecha y adicionalmente podemos establecer otros parámetros que determinan el modo en el cual el *"formateador"* manejará los símbolos que podamos pasarle en el patrón de formato.

La sintaxis normal para inicializar un objeto *SimpleDateFormat* es la siguiente:

```
SimpleDateFormat formateador1 = new SimpleDateFormat("MM/dd/yyyy");
SimpleDateFormat formateador2 = new SimpleDateFormat("MMM, dd");
```

Letter	Date or Time Component	Examples
G	Era designator	AD
y	Year	1996; 96
M	Month in year	July; Jul; 07
w	Week in year	27
W	Week in month	2
D	Day in year	189
d	Day in month	10
F	Day of week in month	2
E	Day in week	Tuesday; Tue
a	Am/pm marker	PM
H	Hour in day (0-23)	0
k	Hour in day (1-24)	24
K	Hour in am/pm (0-11)	0
h	Hour in am/pm (1-12)	12
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	978
z	Time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	-0800

```
import java.text.SimpleDateFormat;
import java.util.*;

class Formateador{

	public static void main(String... args){
		Date ahora = new Date();

		SimpleDateFormat f1 = new SimpleDateFormat("MM/dd/yyyy");
		SimpleDateFormat f2 = new SimpleDateFormat("hh 'en punto'");

		System.out.println("Formato 1: " + f1.format(ahora));
		System.out.println("Formato 2: " + f2.format(ahora));
	}
}
```

5.4.4. La clase NumberFormat

Esta clase proporciona métodos para formatear y analizar cantidades numéricas. Como sucede con *DateFormat*, estamos ante una clase abstracta que proporciona un método estático para crear instancias de sus clases:

- `NumberFormat getInstance()`: obtiene un objeto *NumberFormat* para la localización por defecto.
- `NumberFormat getInstance(Locale area)`: obtiene un objeto NumberFormat para la localización especificada.

También nos podemos encontrar un método equivalente a estos dos `getNumberInstance()`.

Formateo y análisis de números

Podemos formatear números a través del método:

```
String format(Number num)
```

se aplica al número enviado como argumento el formato de la localización asociada al objeto *NumberFormat*.

Si por el contrario queremos analizar un numero a partir de su representación como cadena de caracteres usaremos

```
Number parse(String num)
```

```
import java.text.NumberFormat;
import java.util.Locale;

class FormateoNumber{

	public static void main(String... args){
		double n = 2340.75;

		Locale loc1 = new Locale("us");
		Locale loc2 = new Locale("fr");
		Locale loc3 = new Locale("nl");

		NumberFormat nf1 = NumberFormat.getInstance(loc1);
		NumberFormat nf2 = NumberFormat.getInstance(loc2);
		NumberFormat nf3 = NumberFormat.getInstance(loc3);

		System.out.printf("US: %s\n", nf1.format(n));
		System.out.printf("FR: %s\n", nf2.format(n));
		System.out.printf("NL: %s\n", nf3.format(n));
	}
}
```

Formato de tipo de moneda

Si lo que queremos es aplicar un formato moneda a una cantidad numérica, deberíamos crear un objeto *NumberFormat* a partir del método `getCurrencyInstance()`, en lugar de usar `getInstance()`.

A continuación un ejemplo:

```
import java.text.NumberFormat;
import java.util.Locale;

class Moneda{

	public static void main(String... args){

		double dinero = 12345.6;
		NumberFormat nf;
		Locale loc = new Locale("es","es");
		nf = NumberFormat.getCurrencyInstance(loc);
		System.out.printf("ES €: %s",nf.format(dinero));
	}
}
```

Tabla resumen

Class	Key Instance Creation Options
util.Date	new Date(); new Date(long millisecondsSince010170);
util.Calendar	Calendar.getInstance(); Calendar.getInstance(Locale);

util.Locale	Locale.getDefault();
	new Locale(String language);
	new Locale(String language, String country);

text.DateFormat	DateFormat.getInstance(); DateFormat.getDateInstance(); DateFormat.getDateInstance(style); DateFormat.getDateInstance(style, Locale);
text.NumberFormat	NumberFormat.getInstance() NumberFormat.getInstance(Locale) NumberFormat.getNumberInstance() NumberFormat.getNumberInstance(Locale) NumberFormat.getCurrencyInstance() NumberFormat.getCurrencyInstance(Locale)

5.5. Clases Envoltorio

Para cada uno de los tipos básicos, JAVA proporciona una clase que lo representa, se las conoce como *clases envoltorio*, y sus usos principales son:

- **Encapsular un dato básico en un objeto**: la mayoría de las clases JAVA utilizan métodos para realizar algún tipo de manipulación con datos de tipo objeto. Por ejemplo, el método *add()* de la clase *Vector*, permite almacenar objetos en una colección. En este caso el dato tienen que ser un objeto.

- **Conversión de una cadena a un tipo básico**: en la mayoría de las operaciones de entrada de datos, estos llegan en formato cadena de caracteres. Las clases envoltorio proporciona unos métodos estáticos que permiten convertir una cadena de texto, formada por caracteres, en un tipo numérico.

Las clases envoltorio son: *Byte, Short, Character, Integer, Long, Float, Double y Boolean.* Todas ellas pertenecen al paquete *java.lang.*

5.5.1. Encapsulamiento de un tipo básico

Todas las clases envoltorio permiten crear un objeto de la clase a partir del tipo básico, así por ejemplo:

```
int num = 123;
Integer numInteger = new Integer(num);
```

A excepción de *Character*, las clases envoltorios también permiten crear objetos partiendo de la representación como cadena del dato, por ejemplo:

```
String cad = "4.65";
Float ft = new Float(cad);
```

De cara a recuperar el valor a partir del objeto, las ocho clases de envoltorio proporcionan un método con el formato **xxxValue()** que devuelve el dato encapsulado en el objeto, y donde **xxx** representa el nombre del tipo en el que se quiere obtener el dato, por ejemplo:

```
float dato = ft.floatValue();
int n = numInteger.intValue();
```

5.5.2. Conversión de cadena a tipo numérico

Las clases numéricas proporcionan un método estático **parseXxx(String)** que permite convertir la representación en forma de cadena de un número en el correspondiente tipo numérico, donde **Xxx** es el nombre del tipo al que se va a convertir la cadena de caracteres, en función de la clase que se utilice, por ejemplo:

```
String cad1=”25”, cad2=”12.2”;
int num = Integer.parseInt(cad1);
double num2 = Double.parseDouble(cad2);
```

5.5.3. Autoboxing

Representa otra de las nuevas características del lenguaje incluidas a partir de la versión 5 siendo una de las mas practicas.

Consiste en la encapsulación automática de un dato básico en un objeto de envoltorio, mediante la utilización del operador de asignación, por ejemplo:

```
int n = 123;
Integer num = new Integer(n);
//Con Autoboxing
Integer p = n;
```

Esto quiere decir que la creación del objeto envoltorio se produce implícitamente al asignar el dato a la variable objeto.

De la misma forma, para obtener el dato básico (*autounboxing*) no será necesario recurrir al método **xxxValue()**, esto se realizará implicitamente al utilizar la variable objeto en una expresión, por ejemplo para el caso anterior:

```
int valor = p;
```

```
class AutoB{

      public static void main(String... args){

            Integer []valores = {1,2,3,4,5,6};
            int suma = 0;

            for(Integer nums:valores){
                  suma +=nums;
            }
            System.out.printf("SUMA: %d",suma);
      }
}
```

```
Salida:
```

5.5.4. Resumen métodos clases envoltorio

Los métodos más importantes proporcionados por estas clases son:

- `static objeto_tipo valueOf(String valor)`

Método estático disponible en todas las clases, a excepción de *Character*, que a partir de la representación *String* del tipo devuelve el objeto envoltorio.

- `tipo xxxValue()`

Devuelve el valor envuelto por el objeto en el tipo especificado, siendo xxx el nombre del tipo que se quiero obtener el dato. En el caso de las clase numéricas cada una de ellas dispone del método `xxxValue()` para obtener el valor envuelto por el objeto en cada uno de los tipos numéricos existentes.

- `static tipo parseXxx(String valor)`

Método estático disponible solamente para las clases numéricas que permite convertir la representación en forma de texto del número en el valor numérico correspondiente, siendo Xxx el tipo al que se va a convertir.

- `String toString()`

Disponible en todas las clases envoltorio, y devuelve la representación String del tipo envuelto.

- `static String toString(tipo_numerico valor)`

Las seis clases numéricas disponen de una versión estática del método `toString()` en la que se suministra como parámetro el dato en su tipo básico.

5.6. Entrada y salida en JAVA

Una de las operaciones más habituales es el tener que intercambiar datos con el exterior. Para ello, Java nos proporciona el paquete java.io que incluye una serie de clases que nos permiten gestionar la entrada y salida:

CLASE	OPERACIÓN
PrintStream	Salida
InputStream	
▫ **BufferedRead**	Entrada
▫ **InputStreamReader**	

5.6.1. Salida de datos

El envío de datos al exterior se gestiona al través de la clase *PrintStream*, utilizándose un objeto de la misma para acceder al dispositivo de salida. El proceso de envío se resume en estos pasos:

1. **Obtención del objeto PrintStream:** se debe crear un objeto *PrintStream* asociado al dispositivo de salida, la forma de hacerlo dependerá del dispositivo en cuestión.
2. **Envío de datos al stream:** la clase *PrintStream* dispone de los métodos `print(String cadena)` y `println(String cadena)` para enviar una cadena de caracteres al dispositivo de salida, diferenciándose uno de otro que el segundo añade un salto de línea al final de la cadena.

Salida con formato – El método printf()

A partir de la versión 5, la clase *PrintStream* proporciona un método de escritura que permite aplicar un formato a la cadena de caracteres que se va a enviar a la salida. La sintaxis del método `printf()` es:

```
printf(String formato, Object… datos)
```

donde:

- *formato*: consiste en una cadena de caracteres con las opciones de formato que van a ser aplicadas sobre los datos a imprimir. La sintaxis de esta cadena es:

```
%[pos_argumento$][indicador][minimo][.num_decimales]conversión
```

- *datos*: representa la información que va a ser enviada a la salida y sobre la que se va a aplicar el formato.

```
%[pos_argumento$][indicador][minimo][.num_decimales]conversion
```

El significado de cada uno de estos es:

- ***pos_argumento:*** representa la posición del argumento sobre el que se va a aplicar el formato, siendo el primer argumento el que ocupa la posición 1.
- ***indicador:*** conjunto de caracteres que determinan el formato de salida, destacamos:
  - '-' El resultado aparecerá alineado a la izquierda.
  - '+' El resultado incluirá siempre el signo.
- **minimo:** representa el número mínimo de caracteres que serán representados.
- **num_decimales:** numero de decimales que serán representados, por lo que solo es aplicable a datos de tipo *float* o *double* (cuidado que va precedido de .)
- **conversión:** carácter que indica cómo tiene que ser formateado el argumento, a continuación se presentan los más utilizados:

CARÁCTER	FUNCIÓN
's' 'S'	Si el argumento es *null* se formateará como *null*, en cualquier otro caso se obtendrá el *argumento.toString()*.
'c' 'C'	El resultado será un carácter UNICODE.
'd'	El argumento se formateará como un entero en notación decimal.
'x' 'X'	El argumento se formateará como un entero en notación hexadecimal.
'e' 'E'	El argumento se formateará como un número decimal en notación científica.
'f'	El argumento se formateara como un número decimal.

5.6.2. Entrada de datos

La lectura de datos se gestiona a través de la clase *InputStream* que estará asociada a un dispositivo de entrada (teclado, fichero, etc…).

El método `read()` proporcionado por esta clase para la lectura de datos no nos ofrece la misma potencia que *print* o *println* para la escritura. Una llamada a `read()` devuelve el ultimo carácter introducido a través de dispositivo, esto implica que para leer una cadena carácter sería necesario hacerlo carácter a carácter, lo que haría bastante ineficiente el código.

Por ello es preferible utilizar otra clase del paquete java.io, *BufferedReader*, realizando los siguientes pasos:

1. **Crear un objeto InputStreamReader:** nos permitirá convertir los bytes recuperados del stream de entrada en caracteres. Es necesario indicar el objeto *InputStream* de entrada, si este es el teclado, lo tenemos referenciado por el atributo estático "in" de la clase *System*:

```
InputStreamReader rd;
rd = new InputStreamReader(System.in);
```

2. **Crear un objeto *BufferedReader*:** a partir del objeto anterior construiremos un *BufferedReader* para la lectura de las cadenas que nos lleguen:

```
BufferedReader bf;
bf = new BufferedReader(rd);
```

3. **Invocar al método `readLine()`:** este método nos devuelve todos los caracteres introducidos hasta un salto de línea, si lo utilizamos para leer cadenas de caracteres desde teclado devolverá los caracteres introducidos desde el principio de la línea hasta la pulsación de la tecla *Enter:*

```
String cadena = bf.readLine();
```

```
import java.io.*;

class LecEsc1{

      public static void main(String... args) throws IOException{

            String cadena;
            InputStreamReader ir;
            BufferedReader br;

            ir = new InputStreamReader(System.in);
            br = new BufferedReader(ir);

            System.out.println("Introduce tu nombre: ");
            cadena = br.readLine();

            System.out.println("Hola "+cadena+" buenos dias!!!");
      }
}
```

```
Salida:

¿Y si en lugar del nombre, pedimos la el año de nacimiento para
calcular la edad?
```

5.6.3. La clase Scanner

Hemos visto que utilizando las clases anteriores la lectura de datos puede convertirse en proceso bastante engorroso. Al fin de simplificar este proceso, con la versión 5 de Java se incorpora una clase **`java.util.Scanner`**.

Creación de un objeto Scanner

Para tener acceso a los datos de entrada lo primero que necesitamos es crear un objeto scanner asociado al *InputStream* del dispositivo de entrada. Si es el teclado:

```
Scanner sc = new Scanner(System.in);
```

La cadena de caracteres introducida por teclado hasta la pulsación de la tecla *Enter* es dividida por el objeto scanner en un conjunto de bloques de caracteres de longitud variable, denominados *tokens*. Por defecto, el carácter utilizado como separador de *tokens* es el espacio en blanco, por ejemplo para la cadena "Esto es una cadena":

Token1	Token2	Token3	Token4
Esto	es	un	cadena

Métodos de la clase Scanner

Los métodos más importantes para esta clase son:

Método	Función
String next()	Devuelve el siguiente token.
boolena hasNext()	Indica si existe o no un nuevo token para leer.
xxx nextXxx()	Devuelve el siguiente token como un tipo básico, siendo Xxx el nombre de este tipo básico.
boolean hasNextXxx()	Indica si existe o no un token siguiente del tipo especificado.
void useDelimiter(String d)	Establece un nuevo delimitador

NOTA: Como hemos indicado, el delimitador de token es el espacio en blanco, esto significa que si pedimos el nombre por teclado e introducimos "Juan Carlos" al ejecutar la instrucción:

```
String nombre= sc.next();
```

La cadena recuperada en la variable *nombre* será "Juan", y esto puede implicar también un segundo problema, si a continuación de pedir el nombre por ejemplo pidiéramos la edad, al intentar recogerla con `nextInt()`, ¿Qué ocurriría? Que el programa insertaría en esa variable la segunda parte de la cadena de caracteres del nombre, intentando convertir una cadena de caracteres a un *int* lo que provocará una excepción de tipo `java.util.InputMismatchException`.

Para evitar este problema podemos realizar dos cosas, bien hacemos uso del método `useDelimiter()` para establecer un nuevo delimitador, por ejemplo el retorno de carro "\n. ", o utilizamos el método `readLine()` de la clase Scanner para leer hasta el final de línea. La primera solución sería la más optima si trabajáramos con ficheros y la segundo si estamos leyendo valores introducidos por teclado. Por ejemplo:

```
import java.util.*;

class LecEsc2{

	public static void main(String... args){

		Scanner tec = new Scanner(System.in);
		String nombre;

		System.out.println("Introduce tu nombre: ");
		nombre = tec.nextLine();

		System.out.println("Introduce tu edad: ");
		int edad = tec.nextInt();

		System.out.println("Hola "+nombre+" tienes "+edad+" aNNos");
	}
}
```

```
Salida:
```

PRÁCTICA 5.6.3

5.7. Expresiones regulares

Java pone a disposición del programador paquetes que facilitan el manejo de expresiones regulares. Las clases más importantes para trabajar con expresiones regulares son: *Pattern, Matcher, StringTokenizer* …

Para trabajar con expresiones regulares sencillas usaremos *StringTokenizer*, pero para expresiones más potentes utilizaremos la combinación de *Pattern* y *Matcher*. La clase *Pattern* define el patrón, el cual después mediante *Matcher* crearemos el objeto encargado de indicar si un elemento pertenece a nuestro lenguaje o no.

En primer lugar, para hacer uso de estas librerías hace falta importarlas:

- `import java.util.regex.Matcher;`
- `import java.util.regex.Pattern;`

5.7.1. Definición de un patrón

Un patrón es un objeto de la clase *Pattern* que para crearlo debemos de utilizar un método estático de esta clase llamado `compile()`. La sintaxis sería:

```
static Pattern compile (String reg)
```

donde *reg* representa la expresión regular que define el patrón. Por ejemplo:

```
Pattern pat = Pattern.compile("PAT[AO]");
```

5.7.2. Búsqueda de coincidencias

Una vez definido el patrón, éste se puede aplicar sobre una determinada cadena de caracteres para comprobar si existe una parte de la misma que coincida con los criterios establecidos en la expresión del patrón.

Esta búsqueda de coincidencias se realiza utilizando los métodos de la clase *Matcher*. La clase *Pattern* dispone del método `matcher()` que permite crear un objeto *Matcher* a partir de la cadena donde se desea realizar la búsqueda. Por ejemplo:

```
Matcher mat = pat.matcher("PATO");
```

Solo nos queda comprobar que la cadena ha sido encontrada:

```
if(mat.matches()) {
      System.out.println("ENCONTRADA!!");
}
else {System.out.println("NO ENCONTRADA");}
```

Además del método `matches()` para la comprobación de coincidencias, la clase *Matcher* proporciona otros métodos para la búsqueda de coincidencias, que veremos más adelante, pero antes vamos a estudiar un poco más la sintaxis de las expresiones regulares.

5.7.3. Construir expresiones regulares

Para construir las expresiones regulares utilizas los caracteres de los cuales tenemos diferentes tipos:

- ***Caracteres Literales:*** Cuando un carácter forma parte de la expresión regular, este se incluirá directamente dentro de la expresión regular de búsqueda en la posición donde dicho carácter tenga que aparecer.
  Por ejemplo, en la expresión PAT[AO], contiene los literales P, A y T, indicando que la cadena buscada comienza por los caracteres PAT (cuidado que en este sentido es CASESENSITIVE).

- ***Caracteres Alternativos:*** Para indicar que en una determinada posición puede aparecer cualquiera de los caracteres pertenecientes a un conjunto dado, dicho conjunto tendrá que delimitarse con corchetes []. Por ejemplo:
  - [ABC]: puede parecer una A o B o C.
  - [A-E]: puede aparecer un letra que se encuentre entre la A y la E.
  - [^CD]: cualquier carácter que no sean los indicados.
  - [0-9][a-e][A-E]: ¿?

- ***Caracteres Especiales:*** Son caracteres que tienen un significado dentro de la sintaxis de patrones. Los mas empleados son:

Carácter	Función
.	Representa cualquier carácter.
\d	Digito entre 0 y 9.
\D	Cualquier carácter que no sea un digito.
\s	Espacio en blanco.
\w	Un carácter de palabra (letra o numero).
\W	Cualquier carácter no alfanumérico.
\t	Tabulador

- ***Cuantificadores:*** Son caracteres que determinan la frecuencia con la que pueden aparecer las expresiones asociadas. Tenemos:

Símbolo	Función
+	La expresión puede aparecer una o más veces.
?	La expresión puede aparecer ninguna o una sola vez.
*****	La expresión puede aparecer cualquier número de veces.

Estos caracteres se colocan a continuación de la expresión a la que se quiere aplicar. Por ejemplo, la expresión regular "\d*" representaría un cantidad numérica de cualquier número de cifras.

- ***Grupo de caracteres:*** Para agrupar un conjunto de caracteres en una unidad individual se deberán indicar entre paréntesis (). Por ejemplo "(pato)+" indicando que la palabra *pato* puede aparecer una o más veces seguidas.
Algunos ejemplos:

Expresión	Significado
\d\d/\d\d/\d\d	Formato de fecha corta
\w+\.?\w+@\w+\.\w+	Dirección de correo electrónico
www\..+\.com	Dirección web con dominio .com

5.7.4. Pattern

En una expresión regular existe un patrón, el cual es el encargado de definir mediante una determinada notación un lenguaje. La clase *Pattern* define ese patrón. Los métodos para esta clase son:

METODO	DEFINICIÓN
`static Pattern compile (String expreg)`	Crea un patrón a partir de la expresión regular.
`static Pattern compile(String regex, int flags)`	Crea un patrón a partir de la expresión teniendo en cuenta los flags. Los flags son opciones que se incluyen para tener un trato especial cuando se esté trabajando con la expresión regular. Ejemplo: Distinga entre mayúsculas y minúsculas, tendré que añadir el flag *Pattern.CASE_INSENSITIVE*.
`int flags ()`	Devuelve los flags asociados a la expresión regular o patrón.
`Matcher matcher(CharSequence input)`	Realizará el tratamiento del patrón sobre el texto que se le pase como entrada.
`static boolean matches(String regex, CharSequence input)`	Permite la no utilización de un *matcher* para indicar si una expresión regular puede albergar una cadena o parte de esta.
`String pattern()`	Devuelve la expresión regular asociada al objeto *Pattern* sobre el que estemos trabajando.
`String [] split (CharSequence input)`	Crea un array de String con las diferentes cadenas en las que se ha dividido.

5.7.5. Matcher

Una vez definido el patrón con *Pattern*, tenemos que crearnos un objeto que al recibir una cadena de caracteres analice si dicha cadena o las subcadenas que la componen pertenecen al lenguaje dado.

METODO	DEFINICIÓN
`boolean matches()`	Indica si la cadena de caracteres se ajusta o no al formato definido por la expresión regular.
`boolean find()`	Localiza la siguiente coincidencia con el patrón. Si no hay coincidencias devuelve *false*. Ida: combinar este método con un bucle para iterar sobre una cadena.
`int start()`	Devuelve la posición del primer carácter del trozo de cadena que se ajusta al patrón.
`int end()`	Devuelve la posición del carácter siguiente al último trozo de la cadena que se ajusta al carácter.
`String group()`	Devuelve el trozo de la cadena que se ajusta al patrón.

```
import java.util.regex.*;

class Busqueda1{

 public static void main(String... args){

  String patron = "www\\.\\w*\\.es";

  StringBuilder cadena = new StringBuilder();
  cadena.append("En esta cadena de caracteres estan incluidas las direcciones");
  cadena.append(" uno de los buscadores es www.google.es, aunque tambien");
  cadena.append(" podemos hacer uso de www.msn.es y finalmente si quieres");
  cadena.append(" encontrar trabajo accede a www.infojobs.es");

  Pattern pat = Pattern.compile(patron);
  Matcher mat = pat.matcher(cadena);

  while(mat.find()){
   System.out.printf("%s\n",mat.group());
  }
 }
}
```

Salida:

```
(>java Busqueda1
www.google.es
www.msn.es
www.infojobs.es
```

```
import java.util.regex.*;

class Busqueda2{

	public static void main(String... args){

		String cadena ="ac abc a c";
		String patron ="a.c";

		Pattern pat = Pattern.compile(patron);
		Matcher mat = pat.matcher(cadena);

		while(mat.find()){
			System.out.printf("%d %s\n",mat.start(), mat.group());
		}
	}
}
```

Salida:

```
import java.util.regex.*;

class Busqueda3{

	public static void main(String... args){

		//Cuidado que \d daria error ya que JAVA reserva el \ para
		//escapar caracteres
		String patron1 ="\\d\\w";
		String patron2 ="\\d\\w*";
		String patron3 ="\\d\\w?";
		String patron4 ="\\d\\w+";
		String patron5 ="\\w\\d";
		String cadena ="ab4 56_7ab";

		Pattern pat = Pattern.compile(patron1);
		Matcher mat = pat.matcher(cadena);

		while(mat.find()){
			System.out.printf("%d %s\n",mat.start(), mat.group());
		}
	}
}
```

Salida:

```
class Separar{

	public static void main(String... args){

		String cadena="Juan.Carlos.Gonzalez.Zarza";

		String []cads = cadena.split("\\.");

		for(String items: cads){
			System.out.printf("%s\n",items);
		}
	}
}
```

Salida:

6. Programación Orientada a Objetos con JAVA

En los primeros temas dijimos que Java es un lenguaje totalmente orientado a objetos, esto significa que podemos aplicar en una aplicación Java todas las características y beneficios que proporciona este modelo de programación (de momento no hemos visto programáticamente ninguna).

Llegados a este punto vamos a estudiar todos estos conceptos en los que se basa la POO y su aplicación en Java. Los distintos temas que vamos a estudiar son:

- Empaquetado de clases.
- Modificadores de acceso.
- Encapsulación.
- Sobrecarga de métodos.
- Constructores.
- Herencia.
- Sobrescritura de métodos.
- Clases abstractas e interfaces.
- Polimorfismo.

6.1. Empaquetado de clases

La organización de las clases en paquetes facilita el uso de las mismas en otras clases. Por lo tanto vamos a utilizar esta técnica en el desarrollo de nuestras propias clases, para ello seguiremos los siguientes pasos:

1. **Creación de directorios (carpetas):** un paquete no es más que una carpeta dentro del PC donde estamos desarrollando una aplicación, así pues lo primero que tendremos que hacer es crear estos directorios.
2. **Empaquetado de las clases:** una vez tenemos los directorios, procederemos al empaquetado de las clases, para ello hemos de utilizar la sentencia *package* en el archivo de código fuente de la clase (.java). La sintaxis:

```
package nombre_paquete;
```

NOTA: Esta sentencia **DEBE** de ser la primera instrucción del archivo .java, antes incluso de los import, y afectará a todas las clases existentes en el archivo.

3. **Compilación:** como ya hemos visto en algunos ejemplos anteriores, para compilar una clase Java, nos colocamos en la línea de comandos en cada uno de los subdirectorios y:

```
raiz\paquete>javac Clase.java
```

También se puede invocar al comando *javac* desde el directorio raíz o de trabajo:

```
raiz>javac paquete\Clase.java
```

Ejemplo:

```
package paquete;

public class ClaseIn{

	public String getMensage(){
		return "Hola";
	}
}
```

```
import paquete.ClaseIn;
public class ClaseOut{

	public static void main(String... args){

		ClaseIn ci = new ClaseIn();
		System.out.printf("%s", ci.getMensage());
	}
}
```

6.2. Modificadores de acceso

Aunque no es un punto perteneciente a la programación orientada a objetos puros, es conveniente aclarar su uso. Se utiliza para definir la visibilidad de los miembros de una clase (atributos y métodos) y de la propia clase.

Existen cuatro tipos de modificadores de acceso:

- **private:** aplicable a atributos y métodos, **pero no a clases**, significa que su uso está restringido al interior de la clase, lo que significa que solo se puede utilizar dentro de esta.

- **ninguno (default):** es el acceso por defecto al no especificar nada, aplicable a clase, método y atributo, y significa que únicamente las clases de su mismo paquete tendrán acceso a los mismos.
- **protected:** mas propio de la herencia, aplicable a métodos y atributos, y significa que éstos pueden ser utilizados por cualquier otra clase de su mismo paquete, además, por cualquier subclase de ella independientemente del paquete en el que se encuentre. **No aplicable a nivel de clase.**

- **public:** es el máximo nivel de visibilidad. Un elemento, clase, método o atributo con este tipo de visibilidad es accesible desde cualquier clase independientemente de la clase donde se encuentre.

Cuadro resumen:

	private	**(default)**	**protected**	**public**
Clase	NO	SI	NO	SI
Método	SI	SI	SI	SI
Atributo	SI	SI	SI	SI
Variable local	NO	NO	NO	NO

6.3. Encapsulación

Una clase está compuesta por un lado, de métodos que determinan el comportamiento de los objetos de la clase y, por otro, de atributos que representan las características de los objetos de la clase.

Los métodos que queramos exponer al exterior tendrán que llevar el modificador de acceso *public*, mientras que los atributos **suelen** tener acceso *private*, de modo que solamente puedan ser accesibles desde el interior de la clase.

Esa es precisamente la idea de la encapsulación: mantener los atributos de los objetos como privados y proporcionar acceso a los mismos a través de métodos públicos, ya que nos reportara grandes beneficios, entre los que destacamos:

- Protección de datos sensibles.
- Facilidad y flexibilidad en el mantenimiento de las aplicaciones.

6.3.1. Protección de datos

Imaginemos que queremos crear una clase para representar los Empleados de nuestra Cía. Dicha clase nos proporcionará diversos métodos para trabajar con los empleados, además de disponer de un par de atributos que caracterizaran a la clase, como son el número de empleado y el salario del mismo.

Si dejáramos de lado la encapsulación, nuestra clase se presentaría de la siguiente manera:

```
package beans;

public class Empleado{
	public int numEmpleado;
	public double salario;
}
```

Al intentar utilizar esta clase desde cualquier otro programa e intentar asignar valores a los atributos, nada impedirá al programador que va a realizar esta tarea hacer algo como:

```
Empleado emp = new Empleado();
emp.salario = -123.23;
```

Lógicamente valores negativos para el salario no tienen ningún sentido, además de provocar incoherencias en la ejecución de los diferentes métodos de la clase.

A este hecho se le conoce como *corrupción de datos*, y una forma de evitarlo es proteger los atributos del acceso directo desde el exterior mediante la encapsulación, forzando a que dicho acceso se realice siempre de forma "controlada" a través de métodos de acceso.

Por ejemplo:

```
package beans;

public class Empleado{

	private int numEmpleado;
	private double salario;

	public void setSalario(double salario){
		if(salario > 0){
			this.salario = salario;
		}
	}
	public double getSalario(){
		return this.salario;
	}
	public void setNumEmpleado(int numero){
		if(numero > 0){
			this.numEmpleado = numero;
		}
	}
	public double getNumEmpleado(){
		return this.numEmpleado;
	}
}
```

A estos métodos se les conoce vulgarmente como *getters and setters*. Si nos fijamos se sigue un convenio a la hora de nombrar a estos métodos, y es el siguiente:

```
public tipo_atributo get+Nombre_atributo()
public void set+Nombre_atributo(tipo_atributo valor_nuevo)
```

Ahora si quisiéramos crear objetos Empleado y asignarle después unos valores a los atributos:

```
Empleado emp = new Empleado();
emp.setNumEmpleado(0007270);
emp.setSalario(123.23);
```

En el caso de hacer:

```
emp.setSalario(-123.23);
```

la variable salario permanecería inmutable ya que no puede tomar un valor negativo.

6.3.2. this

El "puntero" (por que apunta, no porque sea un puntero), es una variable especial de *solo lectura"* que nos proporciona Java. Contiene una referencia al objeto en el que se usa dicha variable.

Se utiliza en dos situaciones:

- **Para apuntarse a sí mismo.**
- **Para diferenciar entre las variables locales de un método o constructor, y las variables del objeto.**

```
public class UsoThis{
	private int numEmpleado;
	private String nombre;

	public UsoThis(){
		this("Carlos",7270);
	}

	public UsoThis(String nombre, int numEmpleado){
		this.nombre = nombre;
		this.numEmpleado = numEmpleado;
	}

	public String getNombre(){
		return this.nombre;
	}

	public int getNumEmpleado(){
		return this.numEmpleado;
	}

	public static void main(String... args){

		UsoThis ut = new UsoThis();
		System.out.printf("NOMBRE: %s\n",ut.getNombre());
		System.out.printf("NUMERO: %s",ut.getNumEmpleado());
	}
}
```

```
Salida:
```

PRACTICA 6.3

6.4. Sobrecarga de métodos

Otra de las ventajas de la POO es poder tener en una **misma clase varios métodos con el mismo nombre**, a esto se le conoce como ***sobrecarga*** de métodos.

Aunque no nos hayamos dado cuenta, la sobrecarga de métodos la venimos utilizando desde nuestros primeros ejemplos, ya que el método *print* es un método sobrecargado, ya que tenemos una versión de este método dependiendo el tipo de argumento que le pasemos.

La gran ventaja de la sobrecarga es que, si tenemos varios métodos que van a realizar la misma operación, no necesitamos asignarle un nombre diferente a cada uno de ellos (ocasionaría dificultad y confusión), sino que podemos llamarlos igual a todos ellos.

Para que a un método se le considere "sobrecargado" debe cumplir la siguiente condición: **cada versión del método debe distinguirse de las otras en el numero o tipo de parámetros.** El tipo de devolución puede ser el mismo o no, lo que es indispensable es lo anterior.

```
public class SobrecargaMetodos{

	public void calcularAleatorio(){
		int alea = 0;
		alea = (int)((Math.random()*10)+1);
		System.out.printf("Aleatorio %d\n", alea);
	}

	public void calcularAleatorio(int limS){
		int alea = 0;
		alea = (int)((Math.random()*limS)+1);
		System.out.printf("Aleatorio %d\n", alea);
	}

	public int calcularAleatorio(int limS, int limI){
		int alea = 0;
		alea = (int)((Math.random()*limS)+limI);
		return alea;
	}

	public static void main(String... args){

		SobrecargaMetodos scm = new SobrecargaMetodos();
		scm.calcularAleatorio();
		scm.calcularAleatorio(20);
		int alea = scm.calcularAleatorio(30,20);
		System.out.printf("Aleatorio %d\n", alea);
	}
}
```

6.5. Constructores

6.5.1. Definición y utilidad

Para entender la utilidad de los constructores vamos a partir del ejemplo para la clase Empleado. Esta clase va a representar a los empleados de nuestra empresa y vendrá caracterizada por el numero de empleado y el salario; únicamente tendrá una implementación que será la de mostrar por pantalla la información correspondiente al empleado en cuestión:

```
package beans;

public class Empleado{

	private int numEmpleado;
	private double salario;

	public void setSalario(double salario){
		this.salario = salario;
	}
	public double getSalario(){
		return this.salario;
	}
	public void setNumEmpleado(int numero){
		this.numEmpleado = numero;
	}
	public double getNumEmpleado(){
		return this.numEmpleado;
	}
	public void mostrarInfor(){
		System.out.println("Empleado: "+numEmpleado+" "+salario);
	}
}
```

Ahora si quisiéramos crear un Empleado a partir de esta clase y posteriormente ver su información:

```
Empleado emp = new Empleado();
emp.setNumEmpleado(7270);
emp.setSalario(1000);
emp.mostrarInfo();
```

Como podemos suponer cada vez que queramos crear un Empleado es necesario llamar explícitamente a los métodos `setX()` y `getX()`. Esto además de resultar pesado en el caso de tener muchos atributos, puede dar lugar a olvidos y, por lo tanto, ocasionar que ciertos atributos no tomen valores y se inicialicen implícitamente con valores por defecto. Para evitar estos problemas nace el concepto de ***constructor.***

Un constructor es un "método especial" que es ejecutado en el momento de crear un objeto (hacer una llamada al operador new). También los podemos utilizar para añadir aquellas tareas que deban realizarse en el momento en que se crea un objeto de la clase, como por ejemplo, la inicialización de atributos.

A la hora de crear un constructor tenemos que tener en cuenta las siguientes reglas:

- El nombre del constructor debe ser el mismo que el de la clase (mayúsculas y minúsculas).
- Los constructores **NO** tienen tipo de devolución, ni siquiera *void*.
- Los constructores se pueden sobrecargar, lo que significa que se le aplican las mismas reglas que para los métodos sobrecargados, así por lo tanto, una clase puede tener diferentes constructores para inicializar un objeto con diferentes parámetros.
- Toda clase debe tener al menos un constructor.

Dándole una vuelta de tuerca al constructor anterior, obtendríamos:

```
package beans;

public class Empleado{

	private int numEmpleado;
	private double salario;

	public Empleado(int numEmpleado){
		this.numEmpleado = numEmpleado;
	}
	public Empleado(int numEmpleado, double salario){
		this.numEmpleado = numEmpleado;
		this.salario = salario;
	}

	public void setSalario(double salario){
		this.salario = salario;
	}
	public double getSalario(){
		return this.salario;
	}
	public void setNumEmpleado(int numero){
		this.numEmpleado = numero;
	}
	public double getNumEmpleado(){
		return this.numEmpleado;
	}
	public void mostrarInfor(){
		System.out.println("Empleado: "+numEmpleado+" "+salario);
	}
}
```

Ahora podemos hacer uso de los constructores para hacer de una manera más cómoda la creación de un objeto empleado:

```
Empleado emp1 = new Empleado(7270);
Empleado emp2 = new Empleado(7271,2000);
```

6.5.2. Constructores por defecto

¿Qué pasaría si para la clase anterior intento hacer lo siguiente?:

```
Empleado emp = new Empleado();
```

Antes de responder a esta pregunta, vamos a dar unos detalles mas sobre los constructores. Imaginemos que la clase de la que partimos es aquella en la que no definimos ningún constructor (véase la clase que vio la luz en este punto 6.5), si echamos un ojo a las reglas que hay que cumplir para las clases, la cuarta nos dice:

"Toda clase debe tener al menos un constructor"

¿Estamos incumpliendo esta norma? La respuesta es NO. Toda clase, tiene un constructor por defecto que el compilador coloca en su lugar apropiado sin que nosotros los escribamos explícitamente, a éste se le conoce como *constructor por defecto* o *non-args*. Por lo tanto en nuestra clase se está añadiendo:

```
public Empleado(){}
```

Así por lo tanto, cada que se defina una clase sin constructores, el compilador añadirá uno por defecto sin parámetros y sin código. Y hay que tener que **SOLO SERÁ** añadido en el caso de que nosotros no definamos uno, en el momento en el que nosotros escribamos uno, el constructor por defecto dejará de existir, y si queremos hacer uso de él tendremos que definirlo.

La respuesta a nuestra primera pregunta, seria que en ese caso, el compilador daría un error, ya que estamos intentando hacer uso de un constructor que no está definido. El constructor sin argumentos, es útil que acompañe a otros constructores, ya que puede que un momento determinado solo necesitemos hacer uso de alguna utilidad de la clase sin tener que inicializar un objeto.

PRÁCTICA 6.5

6.6. Herencia

6.6.1. Concepto de herencia

La herencia representa uno de los conceptos más importantes y potentes de la POO.

Podemos expresar la herencia como la capacidad de crear clases que adquieran de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

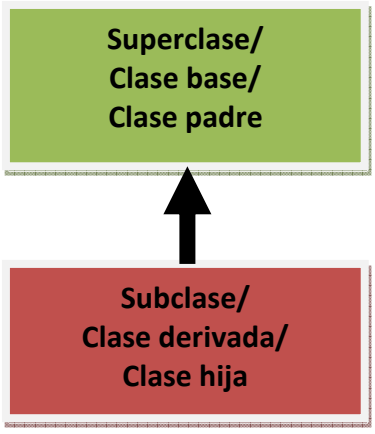
6.6.2. Ventajas de la herencia

Entre las principales ventajas encontramos:

- **Reutilización de código**: en aquellos casos en los que se pretenda crear una clase en la que además de otros métodos tenga unos propios, nos evitamos reescribir todos esos métodos en la nueva clase.
- **Mantenimiento de aplicaciones existentes**: si tenemos una clase con una determinada funcionalidad y tenemos la necesidad de ampliar dicha funcionalidad, no necesitamos modificar la clase existente.

6.6.3. Nomenclatura y reglas

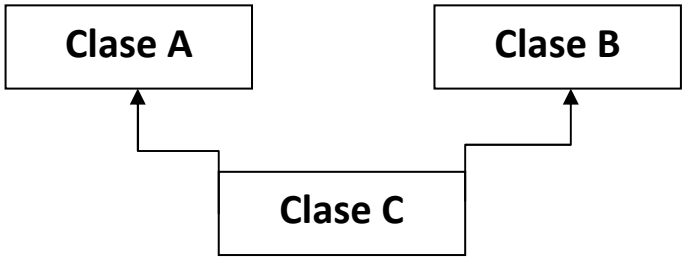
Antes de ver en código como se crean las clases, vamos a definir una nomenclatura básica y conocer ciertas reglas para trabajar con la herencia en Java.



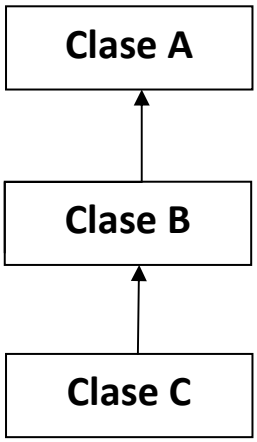
En POO a la clase que va a ser heredada se la conoce como superclase o clase base, mientras que la clase a la que hereda se la conoce como subclase o clase derivada. Gráficamente se representa con una flecha saliendo desde la clase derivada a la clase base.

Hay una serie de reglas sobre la herencia que tenemos que tener en cuenta:

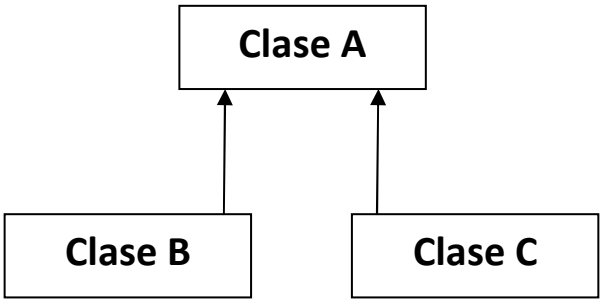
- En Java no está permitida la herencia múltiple, es decir, una subclase no puede heredar de más de una clase.



- Sí es posible una herencia multinivel (es más, en el momento que hagamos nuestra primera herencia estaremos haciendo una de este tipo), es decir, una clase A puede ser heredada por B, y C puede heredar de B.



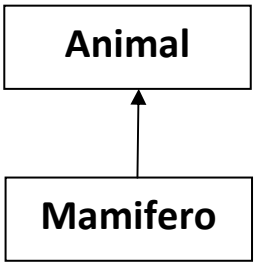
- Una clase puede ser heredada por varias clase (cuidado que pude que nos confunda con el primer caso, pero no es el mismo):



6.6.4. Relación "es un"

La herencia entre dos clases establece una relación entre las mismas del tipo "es un", lo que significa que un objeto de una subclase es un objeto de la superclase (cuidado que lo contrario no es verdad, por eso el orden de la flecha).

Por ejemplo, la clase Vehículo es la superclase de Coche, por lo que podemos decir que, un coche es un vehículo. Esto también es una forma de comprobar que estamos planteando bien la herencia entre las clases de nuestra aplicación:



6.6.5. Creación de la herencia en JAVA

A la hora de definir una clase que va a heredar de otra clase, se utiliza la palabra reservada **extends**, seguida del nombre de la superclase en la cabecera de la declaración:

```
public class sublclase extends superclase{
}
```

La nueva clase podrá incluir atributos y métodos propios para completar su función. Por ejemplo para un caso concreto:

```
public class Jefe extends Empleado{
      private String departamento;
}
```

Todas las clases de Java heredan de alguna clase. En el caso de que no se especifique nada mediante *extends*, significa que la clase va a heredar implícitamente de la clase *Object*.

Aunque una subclase hereda todos los miembros de una superclase, incluido los privados, no tiene acceso a los mismos por la propia definición de *private* (solo accesibles para la propia clase). Así pues únicamente podremos acceder a esos atributos si:

- Tenemos definidos los métodos públicos set()/get() correspondientes.
- Cambiamos la visibilidad, por ejemplo, *protected*.

6.6.6. Ejecución de constructores con la herencia

En la herencia los constructores se comportan de una manera un tanto especial que tenemos que resaltar. Como norma universal tenemos que saber que, **antes de que se ejecute el constructor de una subclase se tiene que ejecutar el constructor de una superclase.**

Por ejemplo: COMPILAR

```
package herencia;

public class Padre{

	public Padre(){
		System.out.println("SOY EL PADRE!!!");
	}
}
```

```
package herencia;

public class Hijo extends Padre{

	public Hijo(){
		System.out.println("SOY EL HIJO!!!!");
	}
}
```

```
package herencia;

public class Principal{

	public static void main(String... args){
		Hijo hi = new Hijo();
	}
}
```

Salida:

```
C>java herencia.Principal
SOY EL PADRE!!!
SOY EL HIJO!!!!
```

La explicación a esta situación la tenemos en el hecho de que el compilador de Java añade, como primera línea de código de todos los constructores de una clase la siguiente instrucción:

```
super();
```

Instrucción que provoca una llamada al constructor sin parámetros de la superclase (los constructores por defecto también incluyen esta instrucción).

Aquellas clases que no hereden también incluirán esta instrucción como primera línea de código en sus constructores, pues que, como hemos dicho anteriormente, toda clase que no herede explícitamente de otra, heredera implícitamente de la clase O*bject.*

Si en vez de llamar al constructor por defecto quisiéramos invocar a cualquier otro constructor de la superclase, deberíamos hacerlo explícitamente añadiendo como *primera línea* del constructor de la subclase la instrucción:

```
super(argumentos)
```

Donde los argumentos son los parámetros que necesita el constructor de la superclase que se desea invocar.

super

El "puntero" *super* es una variable especial que apunta a la superclase de la que estamos heredando. Los usos más comunes de esta variable son:

- Inicializar un objeto a través de un constructor de la superclase de la que estamos heredando.
    - Tiene que ser la primera línea del constructor de la subclase desde la que se está haciendo la llamada.
    - La sintaxis es **`super()`**, importante diferenciarla de la sintaxis en el segundo uso más común.

- Llamar a un método de la superclase de la que estamos heredando como si fuera un método de la clase que hace la llamada.
    - La sintaxis es **super.nombreMetodo**

A continuación ponemos en práctica un ejemplo. Vamos a crearnos tres clases, la primera de ella *SuperPadre.java*¸ tiene un atributo edad que recibe a través de un constructor y que se lo vamos a pasar a través de su clase hija, *SuperHijo.java*. Ambas clases poseen un método para imprimir sus características, con el detalle que la subclase además va a hacer uso del método de la superclase como si fuera suyo.

Finalmente desde la clase *SuperPrincipal.java* iniciamos la aplicación.

```
package herencia.supera;

public class SuperPadre{
	private int edad;
	public SuperPadre(int edad){
		this.edad = edad;
	}
	public String obtenerInfo(){
		return this.edad+"";
	}
}
```

```
package herencia.supera;

public class SuperHijo extends SuperPadre{

	private String nombre;
	public SuperHijo(String nombre){
		//OJO primero la clase padre
		super(27);
		this.nombre = nombre;
	}
	public String obtenerInforHijo(){

		return "Hola "+this.nombre+" tienes " +super.obtenerInfo();
	}
}
```

```
package herencia.supera;

public class SuperPrincipal{

	public static void main(String... args){
		SuperHijo sh = new SuperHijo("Carlos");
		System.out.printf("%s\n",sh.obtenerInforHijo());
	}
}
```

Salida:

```
>javac herencia\supera\SuperPrincipal.java herencia\supera\SuperPadre.java here

>java herencia.supera.SuperPrincipal
Hola Carlos tienes 27
```

MUY IMPORTANTE

Es necesario recalcar que la llamada al constructor de la superclase **debe de ser la primer línea de código del constructor de la subclase,** de no hacerse así se producirá un error de compilación.

PRÁCTICA 6.6.1

6.6.7. Métodos y atributos protegidos

Existe un modificador de acceso pensado para ser utilizado junto la herencia, se trata de *protected*. Un miembro de una clase (atributo o método) definido como *protected* será accesible desde cualquier subclase de esta, independientemente de los paquetes en que estas clase se encuentren. Por ejemplo:

```
package herencia.prot;

public class SuperClasePro{

	private String nombre;
	public SuperClasePro(String nombre){
		this.nombre = nombre;
	}
	protected String getNombre(){
		return this.nombre;
	}
}
```

```
package herencia.supera;
import herencia.prot.SuperClasePro;

public class SuperClaseProHijo extends SuperClasePro{

	private int edad;
	public SuperClaseProHijo(String nombre, int edad){
		super(nombre);
		this.edad = edad;
	}

	public String mostrarDatos(){
		return "Hola "+this.getNombre()+" tienes "+this.edad;
	}
}
```

```
package herencia.prot;
import herencia.supera.SuperClaseProHijo;

public class ClasePrincipalPro{

	public static void main(String... args){
		SuperClaseProHijo scph = new SuperClaseProHijo("Carlos",27);
		System.out.printf("%s\n",scph.mostrarDatos());
	}
}
```

```
Salida:
```

MUY IMPORTANTE

Es importante subrayar que las subclases acceden a los miembros protegidos a través de la herencia, **no pudiendo utilizar una referencia a un objeto de la superclase para acceder al miembro protegido**. Por lo tanto si en el caso anterior hubiéramos intentado:

```
SuperClasePro scp = new SuperClasePro("Carlos");
System.out.printf("%s\n", scp.getNombre()); //ERROR
```

6.6.8. Clases finales

Utilizaremos las clases finales cuando queramos evitar que una clase sea extendida, es decir, la palabra reservada *final*, provoca que sobre una clase no podremos aplicar la herencia.

La sintaxis es como sigue:

```
public final class ClassA{
…
}
```

Si otra clase intentara extender de esta se produciría un error de compilación.

6.6.9. Sobrescritura de métodos

Cuando una clase hereda de otra, el comportamiento de los métodos que hereda no siempre se ajusta a las necesidades de la nueva clase. Por ejemplo, el método *calcularSalario()* dentro de una jerarquía de clases de Empleado no será el mismo si se lo aplicamos a un Jefe/a, a un Becario/a o a un Secretario/a, ya que hay que tener en cuenta otros factores que afectan al cálculo del mismo.

En estos por lo que se opta es por volver a escribir el método heredado, lo que se conoce como sobrescrito de métodos. A la hora de llevar a cabo esta tarea tenemos que tener en cuenta:

- Cuando se sobrescribe un método, este debe tener exactamente el **mismo formato que el método de la superclase** que sobrescribe:
  - Mismo nombre.
  - Mismos parámetros.
  - Mismo tipo (o subtipo) de devolución.

  (NO CONFUNDIR CON LA SOBRECARGA!!!!!!!!)

- El método sobrescrito puede tener un modificador de acceso menos restrictivo que el de la superclase, **pero nunca uno más restrictivo.**
- Para llamar desde el interior de la subclase a la versión original del método de la superclase, debe utilizarse la expresión:

  ¿¿¿¿¿¿¿¿????????

Vamos a ver en un ejemplo práctico la diferencia entre sobrecargar y sobrescribir:

```
package herencia;

public class SobrePadre{

	public void saludar(){
		System.out.println("Hola desde Padre");
	}
}
```

```
package herencia;

public class SobreHijo extends SobrePadre {

	//SOBRESCRITURA
	public void saludar(){
		System.out.println("Hola desde Hijo");
	}

	//SOBRECARGA
	public void saludar(String men){
		System.out.println("Mensaje: "+men);
	}
}
```

PRÁCTICA 6.6.2

6.7. Clases Abstractas

En el siguiente punto nos daremos cuenta del verdadero porque de las clases abstractas ya que estas juegan un papel fundamental dentro del polimorfismo.

6.7.1. Definición

Una clase abstracta es una clase en la que **alguno de sus métodos esta declarado pero no definido**, es decir, se especifica su nombre, sus parámetros y tipo de devolución pero no se incluye código.

¿Cuál es el motivo de declarar un método así? Puede ser que a priori no sepamos cual va a ser el funcionamiento dentro de una jerarquía de clases ya que este será diferente en cada una de ellas.
Atendiendo a esto último podemos decir, vale para esto ya tenemos una solución que es la sobrescritura de métodos, pero y si queremos asegurarnos de que todas las subclases estén obligadas a sobrescribir ese método porque su uso dentro de la aplicación es obligatorio

Obligándonos a cumplir esta última premisa entran en juego las clases abstractas, ya que con estas "obligamos" a que todas las clases respeten el formato del método de la superclase y actúen de dos formas: sobrescribiendo el método o volviéndolo a declarar abstracto.

6.7.2. Sintaxis y características

La sintaxis para la creación de una clase abstracta es la siguiente:

```
public abstract class NombreClase{

      public abstract tipo nombreMetodo(argumentos);
}
```

Nótese que tanto en la declaración de la clase como en la del método abstracto, se debe de utilizar la palabra reservada *abstract*. Se puede también ver como los métodos abstractos son métodos sin cuerpo, su declaración finalizar con un **;** (punto y coma) y dado que no incluye ningún código, no se utilizan las **{ }** (llaves) **.**

Tenemos que tener en cuenta que:

- **Una clase abstracta puede tener métodos no abstractos:** puede incluir tanto métodos abstractos como no abstractos y, por supuesto, atributos. Aunque, basta que tenga un único método abstracto para que la clase tenga que ser declarada como abstracta.
  Y yo me pregunto, ¿Podremos tener una clase abstracta sin métodos abstractos?.

- **No es posible crear objetos de una clase abstracta:** al haber métodos que no están definidos en la clase, no está permitido crear objetos de ella.
  Como ya hemos comentado en la introducción de este punto, el objetivo de las clases abstractas es servir de base a futuras clases que, a través de la herencia, se encargaran de sobrescribir los métodos abstractos y darles sentido.
  La clase abstracta únicamente define el formato que tienen que tener ciertos métodos, dejando a las subclases los detalles de la implementación de los mismos.

- **Las subclases de una clase abstracta no están obligas a sobrescribir todos los métodos abstractos que heredan:** la otra opción es volver a declararlos abstractos, por lo tanto la subclase pasara a ser también abstracta.

```
package herencia.abst;

public abstract class Vehiculo{
      public abstract void arrancar();
}
```

a)

```
package herencia.abst;
public class Coche extends Vehiculo{

      public void arrancar(String men){
            System.out.println("Mensaje: "+men);
      }
}
```

b)

```
package herencia.abst;
public class Coche extends Vehiculo{

      public void arrancar(String men){
            System.out.println("Mensaje: "+men);
      }
      public void arrancar(){
            System.out.println("ARRANCANDO COCHE");
      }
}
```

- **Una clase abstracta puede tener constructores:** esto suena raro raro… Aunque no es posible crear objetos de estas, las clases abstractos serán heredadas por otras clases de que si se podrán crear objetos y, como sabemos, cuando se crear un objeto de una subclase se ejecuta….
  Un ejemplo de esto último sería:

```
public abstract class Figura{

	private String nombre;
	//CONSTRUCTOR CLASE ABSTRACTA
	public Figura(String nombre){
		this.nombre = nombre;
	}
	public String getNombre(){
		return this.nombre;
	}
	//¿CUAL ES EL AREA DE UNA FIGURA?
	public abstract double calcularArea();
}
```

```
package herencia.abst;
public class Triangulo extends Figura{

	private int base, altura;
	public Triangulo(String nombre, int base, int altura){
		super(nombre);
		this.base = base;
		this.altura = altura;
	}

	public double calcularArea(){
		return base*altura/2;
	}

	public int getBase(){
		return base;
	}
	public int getAltura(){
		return altura;
	}
}
```

```
package herencia.abst;
public class PrincipalFiguras{

 public static void main(String... args){
   Triangulo tri = new Triangulo("Triangulo", 12, 12);
   double resultado = tri.calcularArea();
   System.out.println("El area de: "+tri.getNombre()+" vale:
   "+resultado);
 }
}
```

6.8. Polimorfismo

El polimorfismo se basa en gran medida en los conceptos que hemos visto anteriormente, de hecho, es una de las principales aplicaciones de la herencia y supone el principal motivo de la existencia de las clases abstractas.

6.8.1. Asignación de objetos a variables de su superclase

En Java, es posible asignar un objeto de una clase a una variable de su superclase. Esto es aplicable, incluso cuando la superclase es abstracta. Por ejemplo, para el ejercicio anterior podríamos, primero definir una variable del tipo:

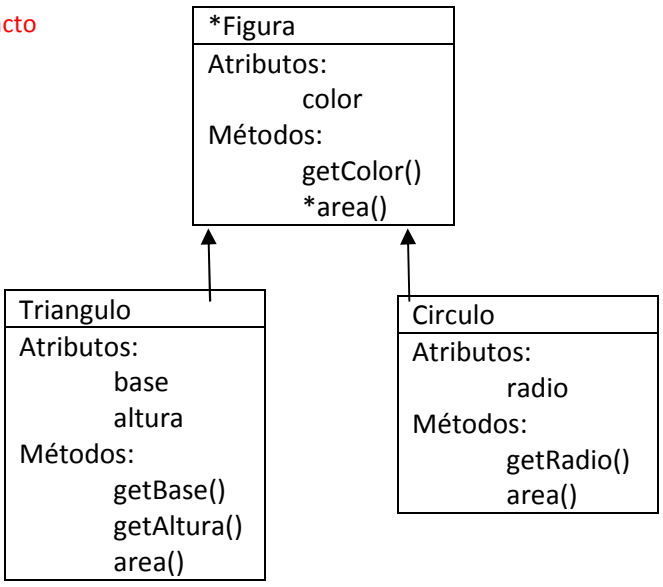
```
Figura f;
```

Ahora sería posible asignar a esta variable objetos de sus subclases, como:

```
f = new Triangulo(…);
```

A partir de aquí, ***se puede utilizar esta variable para invocar a aquellos métodos del objeto que también estén definidos o declarados en la superclase,*** pero no a aquellos que solo existan en la clase a la que pertenece el objeto. Para la jerarquía:

(*) Clase/Metodo Abstracto



Podemos ejecutar:

```
f.getColor();      ¿?
f.area();          ¿?
f.getBase();       ¿?
f.getAltura();     ¿?
```

6.8.2. Definición de polimorfismo

La pregunta más lógica que se nos viene a la cabeza, sería ¿qué utilidad tiene asignar un objeto a una variable de su superclase para llamar a sus métodos, cuando eso mismo lo podemos hacer si le asignamos una variable de su propia clase?

Para responder a esta pregunta quizás lo más fácil sea volver al ejemplo práctico, en el que además vamos a añadir una subclase más de Figura, que será la clase Rectangulo. Según lo dicho en el apartado anterior, podríamos hacer:

```
//Lo primero una variable de la superclase
Figura f;
f = new Triangulo(…);
f = area()//Area del Triangulo
f = new Circulo(…);
f = area()//Area del Circulo
f = new Rectangulo(…);
f = area()//Area del Rectangulo
```

De todo este código se deduce un hecho muy interesante: la misma instrucción *f.area()* permite llamar a distintos métodos *area()*, dependiendo del objeto almacenado en la variable *f.*

Y es en esto en lo que consiste básicamente el polimorfismo, que lo podemos definir como: "**La posibilidad de utilizar una misma expresión para invocar a diferentes versiones de un mismo método**, determinando en tiempo de ejecución la versión del método que se debe ejecutar".

6.8.3. Ventajas del polimorfismo

La principal ventaja que éste ofrece es la **reutilización de código**. Utilizando una variable de una clase puede escribirse una única instrucción que sirva para invocar a diferentes versiones del mismo método, permitiéndose agrupar instrucciones de este tipo en un bloque de código para que pueda ser ejecutado con cualquier objeto de las subclases.

6.8.4. Tipos de retorno covariantes

Al hablar de sobrescritura de métodos en una subclase, dijimos que una de las condiciones que debe cumplir la nueva versión del método era la de mantener invariable el tipo de retorno por el método original.

A partir de la versión Java 5 es posible modificar el tipo de retorno al sobrescribir un método, siempre y cuando el **nuevo tipo sea un subtipo (subclase) del original**. Por ejemplo, supongamos que la clase Figura utilizada en los ejemplos anteriores tuviera declarado un método abstracto *getNewFigura()* que lo único que hace es devolver una copia del propio objeto Figura:

```
abstract Figura getNewFigura();
```

Las subclases Rectangulo, Triangulo y Circulo, al heredar de Figura, dispondrán también de este método y por tanto estarán obligados a sobrescribirlo. En el caso de Circulo, por ejemplo, si no estuviéramos en Java 5 o superior tendríamos:

```
public Figura getNewFigura(){
      return new Circulo(radio,getColor());
}
```

Y lo recogeríamos de tal forma:

```
Circulo cir2 = (Circulo)cir.getNewFigura();
```

Sin embargo, a partir de la versión 5 de Java, todo cambiaria, primero sobrescribir el método seria:

```
public Circulo getNewFigura(){
      return new Circulo(radio,getColor());
}
```

Y para recoger los valores haríamos:

```
Circulo cir2 = cir.getNewFigura();
```

Esto elimina la complejidad de realizar conversiones explicitas a la hora de obtener nuevas copias de objetos de tipo Circulo.

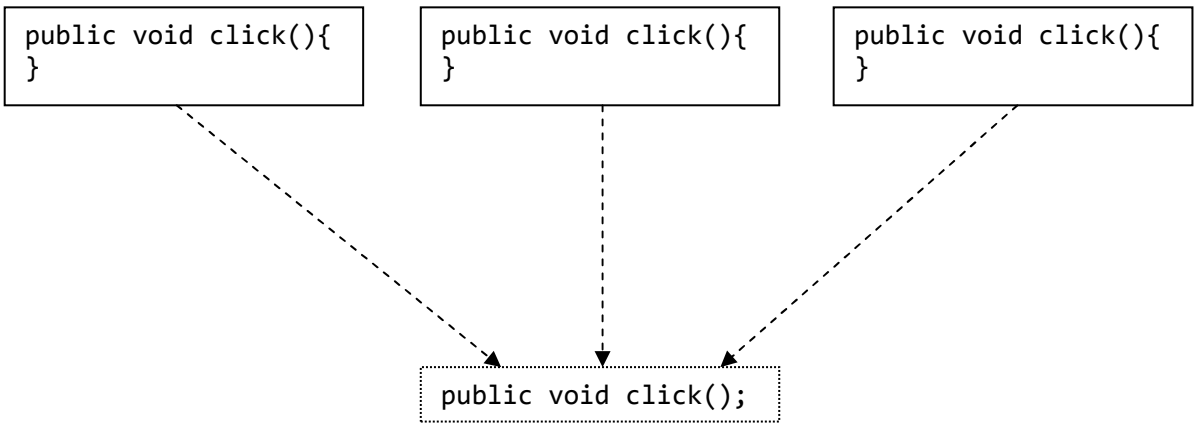
PRÁCTICA 6.8

6.9. Interfaces

Estrictamente hablando, una interfaz es un conjunto de métodos abstractos y de constantes publicas definidas en un archivo .java. Una interfaz es similar a una clase abstracta llevada al límite, es decir, todos sus métodos son abstractos.

La finalidad de una interfaz es definir el formato que van a tener determinados métodos que van a **implementar** ciertas clases.

Graficamente:



Por ejemplo, si quisiéramos gestionar los eventos de una aplicación basada en un entorno gráfico, las clases donde se capturan estos eventos deben de codificar una serie de métodos que se van a ejecutar al producirse estos eventos. Cada tipo de evento tendrá su propio método de respuesta, cuyo formato **está definido en la interfaz.** Así aquellas clases que deseen responder a un determinado evento deberán implementar el método de respuesta de acuerdo al formato definido en la interfaz.

Tenemos que insistir en el hecho de que la interfaz ***no establece lo que un método tiene que hacer y cómo hacerlo, sino en el formato*** (nombre, parámetros, tipo de devolución) ***que este debe tener.***

6.9.1. Definición de una interfaz

Una interfaz se define mediante la palabra reservada ***interface***, utilizando la siguiente sintaxis:

`[public] interface Nombre_Interfaz{`
`  tipo metodo1(argumentos);`
`  tipo metodo2(argumentos);`
`}`
	`public interface Operaciones{`
	`  void rotar();`
	`  String serializar();`
	`}`

Al igual que las clases, las interfaces se definen en archivos .java y, como sucede en las clases, si la interfaz utiliza un acceso *public*, el nombre de la interfaz deberá de coincidir con el fichero .java donde se almacena.

A la hora de crear una interfaz hay que tener en cuenta las siguientes consideraciones:

- **Todos los métodos definidos en una interfaz son públicos y abstractos**, aunque no se indique implícitamente. El uso de los modificadores *abstract* y *public* en la definición de los métodos de la interfaz es, por tanto, redundante si bien su uso no provoca ningún tipo de error.

- **En una interfaz es posible definir constantes.** Además de métodos, las interfaces pueden contener constantes, las cuales son, implícitamente, *public* y *static*. De hecho, tanto los modificadores *public, static* como *final* se pueden omitir en la definición de constantes dentro de una interfaz. Así los siguientes ejemplos serian validos:

```
int k = 23;
public String s = “Hola”;
public static final doublé pi = 3.14;
Object o = new Object();
```

- **Una interfaz no es una clase.** Solo puede contener métodos abstractos y constantes. No puede contener métodos con código, constructores o variables y, por supuesto, no es posible crear objetos de una interfaz.

6.9.2. Implementación de una interfaz

Como ya hemos dicho anteriormente, el objetivo de una interfaz es el de proporcionar un formato común de métodos de clase. Para forzar a que una clase defina el código para los métodos declarados en una determinada interfaz, dicha clase deberá ***implementar*** la interfaz.

En la definición de una clase, se utiliza la palabra *implements* para indicar que interfaz se ha de implementar:

```
public class MiClase implements MiInterfaz{
}
```

```
Por ejemplo:
      public class Triangulo implements Operaciones{
            public void rotar(){
                  //implement el método rotar
            }
            public String Serializar(){
                  //implement el método serializar
            }
      }
```

Sobre la implementación de las interfaces tenemos que tener en cuenta:

- Al igual que sucede al heredar una clase abstracta, **cuando una clase implementa una interfaz, está obligada a definir el código (implementar) de todos los métodos existentes en la misma**. De no ser así, la clase deberá ser declarada abstracta.

- **Una clase puede implementar mas de una interfaz**, en cuyo caso, deberá implementar los métodos existentes en todas las interfaces. El formato será:

  ```
  public class MiClase implements Interfaz1, Interfaz2…{
  }
  ```

- **Una clase puede heredar de otra clase e implementar al mismo tiempo una o varias interfaces**. El implementar una interfaz no impide que la clase puede heredar las características y capacidades de otras clases.
  La sintaxis utilizada para heredar una clase e implementar una interface es:

  ```
  public class MiClas extends SuperC implements Inter1, Inter2{
  }
  ```

- **Una interfaz puede heredar de otras interfaces.** No se trata de una herencia realmente ya que lo que está ocurriendo es que la "subinterfaz" adquiere el conjunto de métodos abstractos existentes.

  ```
  public interface MiInterface extends Interface1, Interface2{
  }
  ```

6.9.3. Interfaces y polimorfismo

Una *variable de tipo interfaz puede almacenar cualquier objeto de las clases que la implementan*, pudiendo utilizar esta variable para invocar a los métodos del objeto que han sido declarados en la interfaz e implementados en la clase; En nuestro caso:

```
Operaciones ope = new Triangulo();
ope.rotar();
ope.serializar();
```

Esta capacidad, unido a su flexibilidad, hace de las interfaces una estructura realmente útil de programación.

6.9.4. Interfaces en J2SE

Además de clases, los paquetes estándar de JAVA incluyen numerosas interfaces, algunas son implementadas por las propias clases de J2SE y otras están diseñadas para ser implementadas en las aplicaciones. Las más importantes son:

- **java.lang.Runnable**: contiene un método para ser implementado por aquellas aplicaciones que a funcionar en modo multitarea.
- **java.util.Enumeration**: proporciona métodos que son implementados por los objetos utilizados para recorres las colecciones.
- **java.awt.event.WindowListener**: proporciona métodos que deben ser implementados por las clases que van a gestionar eventos (clases manejadoras) producidos en la ventana, dentro de una aplicación basada en un entorno gráfico.
- **java.sql.Connection**: interfaz que implementaremos para manejar las conexiones con bases de datos.
- **java.io.Serializable**: no contiene ningún método que deba ser definido por las clases que la implementan, sin embargo, la JVM requiere que dicha interfaz deba ser implementada por aquellas clases cuyos objetos tengan que ser transferidos a algún dispositivo de almacenamiento, como por ejemplo, un archivo de disco.

PRÁCTICA 6.9

7. Colecciones y Tipos Genéricos

7.1. Object

Entre los métodos más importantes a sobrescribir de la clase *Object* tenemos:

- `String toString ( )`
- `boolean equals (Object obj)`
- `int hashcode()`
- `void finalize()`
- `final void notify()`
- `final void notifyAll()`
- `final void wait()`

7.1.1. Sobrescribir toString( )

Todos los objetos tienen un método *toString* que se llama automáticamente cuando el objeto se representa como un valor de texto o cuando un objeto se referencia de tal manera que se espera una cadena.

Por defecto, el método toString es heredado por todos los objetos que descienden de *Object*. Si este método no se sobrescribe en el objeto personalizado, *toString* devuelve [object *type*], donde type es el tipo de objeto. El siguiente código ilustra esto:

```
package colec;
public class SobresString{

	private String cad1,cad2,cad3;
	public SobresString(String cad1, String cad2, String cad3){
		this.cad1 = cad1;
		this.cad2 = cad2;
		this.cad3 = cad3;
	}
	public static void main(String... args){
		SobresString sob = new SobresString("aa","bb","cc");
		System.out.println(sob.toString());
	}
}
Salida: colec.SobresString@10b62c9
```

Para que la salida se correspondiera con lo que nosotros queremos, es decir, con el contenido verdadero del objeto, tendremos que sobrescribir el método toString, por ejemplo, de la siguiente manera:

```
	public String toString(){
		return this.cad1 + " " + this.cad2+ " " +this.cad3;
	}
```

```
Salida: aa bb cc
```

7.1.2. Sobrescribir equals( )

Igual que para el caso de *toString()*, sino se sobrescribe este método se heredara el del clase *Object*, según la cual, la comparación de dos referencias devolverá *true* solamente si ambas apuntan al mismo objeto. Naturalmente esto en todos los casos no es lo que vamos a buscar.

Por ejemplo imaginemos que tenemos un mapa (clase *Map*), este se basa en la utilización de claves únicas, los objetos de clases que no sobrescriben el método *equals()* no podrán ser utilizados como claves.

A la hora de sobrescribir *equals()* el programador decide que se entiende por igual, sin embargo, es conveniente que se **compruebe primero que el objeto pasado como parámetro es una instancia de la clase a la que pertenece el objeto con el que se compara**:

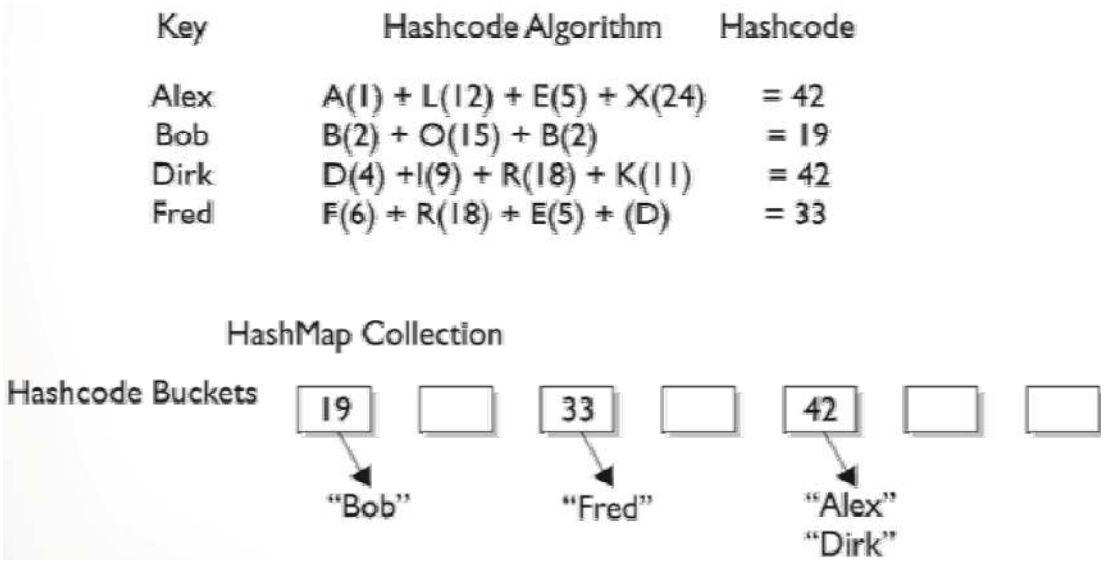
```
public boolean equals(Object obj){
      if((obj instanceof Clase) && …)
}
```

Por otro lado, independientemente de cuál sea el criterio de igualdad que se defina, la sobrescritura de este método debe cumplir las siguientes propiedades:

- **Reflexividad**: `a.equals(a)` debe de ser true.
- **Simetría**: `a.equals(b)` debe de dar el mismo resultado que `b.equals(a)`.
- **Transitividad**: si `a.equals(b)` es true y `b.equals(c)` el true, entonces `a.equals(c)` tiene que ser true.
- Dada una referencia "a" no nula, **la expresión `a.equals(null)` debe devolver false**. Si a es null, se producirá una excepción de tipo NullPointerException.

7.1.3. Sobrescribir hashCode( )

Este método devuelve el valor numérico asociado al objeto. Si se sobrescribe *equals()* también se debe de sobrescribir *hashCode()*, teniendo en cuenta que: **dos objetos considerados iguales utilizando el método *equals()* deben tener idénticos valores de *hashCode()***. Ojo, que lo contrario no tiene porque ser cierto.



El valor *hashcode* de un objeto es utilizado por algunas colecciones (las que tienen el prefijo hash en su nombre) para saber en qué lugar interno debe colocar el objeto.

Al sobrescribir el método *hashCode()* tenemos que tener en cuenta que este deberá cumplir las siguientes propiedades:

- El valor devuelto por *hashCode()* no tiene por qué ser el mismo de una ejecución a otro de la misma aplicación.
- Si dos objetos son iguales de acuerdo a *equals()*, la llamada a sus métodos *hashCode()* debe dar el mismo resultado.

No es obligatorio que dos objetos diferentes tengan diferente hashcode, aunque es conveniente que así sea de cara a mejorar el rendimiento de las colecciones de tipo hash. Tenemos que recordar:

CONDICIÓN	REQUISITO	PERMITIDO (no requerido)
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y) == true`
`x.equals(y) == false`		`Sin requerimientos de hashCode().`
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

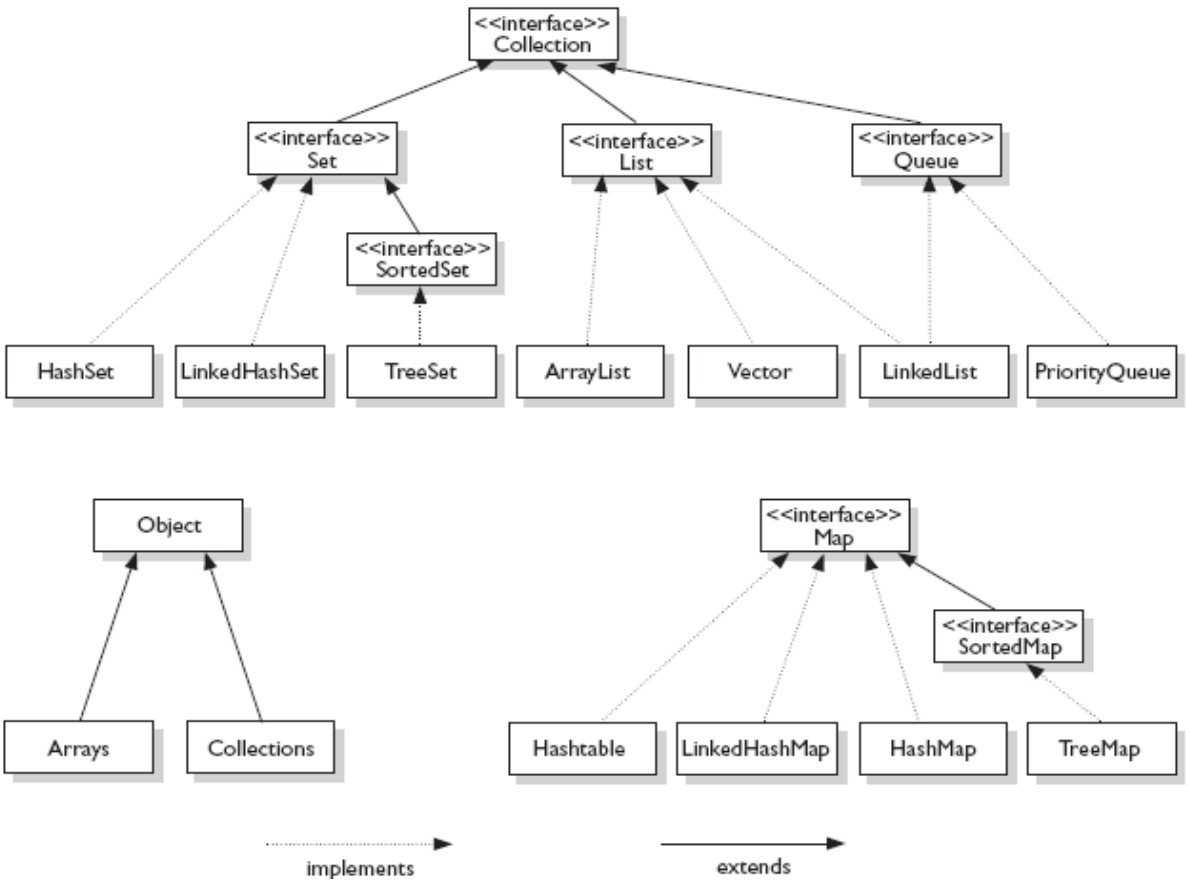
7.2. Colecciones

Una colección es un objeto que almacena un conjunto de referencias a otros objetos, dicho de otra manera, es una especie de array de objetos. A diferencia de los arrays las colecciones son dinámicas, es decir, no tienen un tamaño fijo y permiten añadir y eliminar objetos en tiempo de compilación.

7.2.1. Clases e Interfaces de colección

Las interfaces básicas de colección incluidas en el paquete java.util son:

- **List**: representa listas de objetos en las que cada elemento tiene asociado un índice. Los elementos pueden ser duplicados
- **Set**: representa conjuntos de elementos. No se admiten duplicados.
- **Map**: representa conjuntos de objetos con identificador (clave) único.
- **Queue**: se trata de una interfaz para colecciones de tipo cola (LIFO) y pila (FIFO).



A la hora de trabajar con colecciones, lo más complicado de todo es identificar en cada caso dentro de nuestra aplicación cuál de ellas tenemos que usar para sacar un mayor beneficio al proceso que estamos realizando. Para ayudarnos en esta labor a continuación vamos a describir cada una de las cuales están presentes en el siguiente cuadro:

Lists	Sets	Maps	Queues
ArrayList	HashSet	HashMap	PriorityQueue
Vector	LinkedHashSet	Hashtable	
LinkedSet	TreeSet	TreeMap	
		LinkedHashMap	

7.2.2. Collection

La interfaz más importante es *Collection*. Una *Collection* es todo aquello que se puede recorrer (o "iterar") y de lo que se puede saber el tamaño. Muchas otras clases extenderán *Collection* imponiendo más restricciones y dando más funcionalidades. Es de notar que el requisito de "que se sepa el tamaño" hace inconveniente utilizar estas clases con colecciones de objetos de las que no se sepa "a priori" la cantidad.

Las operaciones básicas de una collection entonces son:

MÉTODO	DEFINICIÓN
`add(T)`	añade un elemento.
`iterator()`	obtiene un "iterador" que permite recorrer la colección visitando cada elemento una vez.
`size()`	obtiene la cantidad de elementos que esta colección almacena.
`contains(t)`	pregunta si el elemento t ya está dentro de la colección.

Tenemos que tener cuidado con los objetos de tipo *Collection*, ya que:

- No se puede asumir que el orden en el que se recorre sea relevante, es decir, que si lo recorremos de nuevo, los elementos puede que aparezcan en un orden diferente al de la primera vez.
- Tampoco se puede asumir que no hay duplicados.
- No se pueden asumir características de rendimiento, es decir, preguntar si existe un objeto en la colección puede tardar mucho tiempo.

Hemos dicho que una de las posibilidades que no ofrece el objeto *Collection* es la de ser recorrido y como a este nivel no está definido un orden, la única manera es proveyendo un iterador, mediante el *método iterator()*.

Un iterador es un objeto "paseador" que nos permite ir obteniendo todos los objetos al ir invocando progresivamente su método *next()*. También, si la colección es modificable, podemos remover un objeto durante el recorrido mediante el método *remove()* del iterador. El siguiente ejemplo recorre una colección de Integer borrando todos los ceros:

```
package colec;

public void borrarCeros(Collection<Integer> ceros){
	Iterator<Integer> it = ceros.iterator();
	while(it.hasNext()){
		int i = it.next();
		if(i == 0){
			it.remove();
		}
	}
}
```

```
¿Alguna solución más rápida?
```

NOTA

Es fácil confundir "*Collection*" con "*Collections*", y viceversa. *Collections* es una clase, con métodos utilitarios. Mientras que *Collection* es una interfaz con declaraciones de métodos comunes a la mayoría de las colecciones, incluyendo *add()*, *remove()*, *contains(), size() e iterator():*

- *Collection* ('C' mayúscula), es la interfaz `java.util.Collection`, la cual extienden Set, List y Queue (no existen implementaciones directas de Collection).

- *Collections* ('C' mayúscula, termina en 's'), es la clase `java.util.Collections`, que contiene gran cantidad de métodos estáticos para utilizar en colecciones.

7.2.3. Interfaz List

Un *List*, o simplemente lista, es una *Collection*. La diferencia es que la lista mantiene un orden arbitrario de los elementos y permite acceder a los elementos por orden. Podríamos decir que en una lista, por lo general, el orden es dato, es decir, el orden es información importante que la lista también nos está almacenando.

No hay ningún método en *Collection* para obtener el tercer elemento. No lo puede haber porque, como se dijo, a nivel *Collection* ni siquiera estamos seguros de que si volvemos a recorrer la colección los elementos aparecerán en el mismo orden. Una lista sí debe permitir acceder al tercer elemento, por eso se añaden los siguientes métodos:

MÉTODO	DEFINICIÓN
`get(int i)`	Obtiene el elemento en la posición *i*.
`set(int i, T t)`	Pone al elemento *t* en la posición *i*.

Existen varias implementaciones para esta interfaz: ArrayList, LinkedList y Vector. Vamos a ver las diferencias que existen entre éstas.

La implementación entre *ArrayList* y *Vector* son prácticamente iguales excepto que los métodos de Vector están sincronizados, por lo tanto, si no vamos a utilizar *Threads* lo mas recomendable (lo más rápido también) es que utilicemos *ArrayList*, así nosotros nos vamos a centrar en *ArrayList* y *LinkedList*.

La ventaja de ***ArrayList*** sobre un array común es que es expansible, es decir que crece a medida que se le añaden elementos (mientras que el tamaño de un array es fijo desde su creación).

- Lo bueno es que el tiempo de acceso a un elemento en particular es ínfimo.
- Lo malo es que si queremos eliminar un elemento del principio, o del medio, la clase debe mover todos los que le siguen a la posición anterior, para "tapar" el agujero que deja el elemento removido. Esto hace que sacar elementos del medio o del principio sea costoso.

La otra implementación es **LinkedList** (lista enlazada). En ésta, los elementos son mantenidos en una serie de nodos atados entre sí como eslabones de una cadena. Cada uno de estos nodos *apunta a su antecesor y al elemento que le sigue*.
No hay nada en cada uno de esos nodos que tenga algo que ver con la posición en la lista.

Para obtener el elemento número "n", esta implementación de *List* necesita entonces empezar desde el comienzo, desde el primer nodo, e ir avanzando al "siguiente" n veces. Buscar el elemento 100 entonces implica 100 de esos pasitos.

- La ventaja es que es posible eliminar elementos del principio de la lista y del medio de manera muy eficiente. Para eliminar un elemento solamente hay que modificar a sus dos "vecinos" para que se "conecten" entre sí ignorando al elemento que se está borrando. Como en una cadena, se retira un eslabón abriendo los eslabones adyacentes al que se elimina y cerrándolos de modo que lo excluyan. No es necesario hacerle ningún cambio al resto de los elementos de la lista.

- Si en otros lenguajes lidiar con listas enlazadas puede ser un poco más trabajoso. En Java, *LinkedList* se usa exactamente igual que otros tipos de *List*, por lo que no hay que saber nada adicional para empezar a usarla. Bueno, esto no es del todo cierto... hay que tener muy en claro sus particularidades en cuanto a rendimiento:

    - Su método `get(int)` es particularmente lento porque, como mencionamos antes, necesita recorrer "n" elementos para llegar al pedido. Esto hace que recorrer la lista con un simple `for` sea tremendamente lento, y la complejidad pasa de ser lineal a cuadrática, es decir, si se recorre así una lista de 300 elementos, se tarda como si tuviera 44.850 elementos!(Un *LinkedList* sólo debe recorrerse mediante iteradores).

Un uso ideal de *LinkedList* es para la creación de "colas", en las que los elementos se añaden al final, y se eliminan del comienzo. Para este uso se puede usar, en vez de *List*, la interfaz Queue (también implementada por *LinkedList*) que es más específica para esta tarea.

De entre los métodos comunes a las clases `ArrayList<E>`, `Vector<E>`, y `LinkedList<E>` los más interesantes son los siguientes:

MÉTODO	DEFINICIÓN
`boolean add(E o)`	Añade un nuevo elemento al final de la colección.
`boolean add(int index, E element)`	Añade un nuevo elemento en la posición especificada.
`boolean addAll(Collection<? extends E> c)`	Añade todos los elementos de la colección especificada a esta colección.
`void clear()`	Elimina todos los elementos de la colección.
`boolean contains(Object o)`	Comprueba si el elemento especificado es parte de la colección.
`E get(int index)`	Recupera el elemento que se encuentra en la posición especificada.
`int indexOf(Object o)`	Devuelve la primera posición en la que se encuentra el elemento especificado en la colección, o -1 si no se encuentra.
`int lastIndexOf(Object o)`	Devuelve la última posición en la que se encuentra el elemento especificado en la colección, o -1 si no se encuentra.
`E remove(int index)`	Elimina el elemento de la posición indicada.
`boolean remove(Object o)`	Elimina la primera ocurrencia del elemento indicado. Si se encontró y se borró el elemento, devuelve true, en caso contrario, false.
`E set(int index, E element)`	Reemplaza el elemento que se encuentra en la posición indicada por el elemento pasado como parámetro. Devuelve el elemento que se encontraba en dicha posición anteriormente.
`int size()`	Devuelve el número de elementos que se encuentran actualmente en la colección.

7.2.4. Interfaz Set

Un *Set* es una *Collection*, que traducimos como un "conjunto", y que los desarrolladores de Java estaban pensando en lo que matemáticamente se conoce como conjunto. Un *Set* agrega una sola restricción: **No puede haber duplicados.**

Por lo general en un *Set* el orden no es dato. Si bien es posible que existan *Set* que nos aseguren un orden determinado cuando los recorremos, ese orden no es arbitrario y decidido por nosotros, ya que la interfaz *Set* no tiene ninguna funcionalidad para manipularlo (como si lo admite la interfaz *List*).

- La ventaja de utilizar *Set* es que preguntar si un elemento ya está en el contenido mediante `contains()` suele ser muy eficiente. Entonces es conveniente utilizarlos cada vez que necesitemos una colección en la que no importe el orden, pero que necesitemos preguntar si un elemento está o no. Como, a diferencia de *Collection*, el orden no necesariamente es preservado, no existen métodos para "obtener el primer elemento".

Las diferentes implementaciones que existen para esta interfaz son: *HashSet*, *LinkedHashSet* y *TreeSet* que vamos a ver a continuación.

HashSet

Existen varias implementaciones de *Set*. La más comúnmente usada es *HashSet*. Los *Set* (y los *Map*, que veremos mas adelante) aprovechan una característica de Java: *Todos los objetos heredan de Object*, por lo tanto todos los métodos de la clase Object están presentes en todos los objetos. Dicho de otra manera, hay ciertas cosas que todo objeto en Java sabe hacer. Éstas son:

- Saber si es igual a otro, con su método `equals()`.
- Devolver un número entero de modo tal que si dos objetos son iguales ese número también lo será (se conoce esto como un *hash*). Esto todo objeto lo sabe hacer con su método `hashCode()`.

La clase *HashSet* aprovecha la segunda de las funciones. A cada objeto que se añade a la colección se le pide que calcule su "hash". Este valor será un número entre -2147483647 y 2147483648. Basado en ese valor se lo guarda en una tabla. Más tarde, cuando se pregunta con `contains()` si un objeto x ya está, habrá que saber si está en esa tabla. ¿En qué posición de la tabla está? *HashSet* puede saberlo, ya que para un objeto determinado, el hash siempre va a tener el mismo valor.

Entonces la función *contains* de *HashSet* saca el *hash* del objeto que le pasan y va con eso a la tabla. En la posición de la tabla hay una lista de objetos que tienen ese valor de *hash*, y si uno de esos es el buscado contains devuelve `true`.

Un efecto de este algoritmo es que el orden en el que aparecen los objetos al recorrer el *Set* es impredecible. También es importante darse cuenta de que es crítico que la función *hashCode()* tiene que devolver siempre el mismo valor para los objetos que se consideran iguales (o sea que *equals()* da `true`). Si esto no es así, *HashSet* pondrá al objeto en una posición distinta en la tabla que la que más adelante consultará cuando se llame a *contains*, y entonces *contains* dará siempre `false`, por más que se haya hecho correctamente el `add()`. Esto mismo puede suceder si se usan como claves objetos que varíen.

TreeSet

Antes de entrar en la descripción de *TreeSet* vaya una breve explicación. Otra cosa que pueden saber hacer los objetos con independencia de cómo y dónde son usados es saber ordenarse. A diferencia de "equals" y "hashCode", que están en todos los objetos, la capacidad de "ordenarse" está sólo en aquellos que implementan la interfaz *Comparable*.

Al implementar esta interfaz promete saber compararse con otros (con el método `compareTo()`), y responder con este método si él está antes, después o es igual al objeto que se le pasa como parámetro. Al orden resultante de usar este método se le llama en Java *"orden natural"*. Muchas de las clases de Java implementan *Comparable*, por ejemplo *String* lo hace, definiendo un orden natural de los *Strings* que es el obvio, el alfabético.

Por ejemplo si tenemos una clase Alumno, queda a nuestro cargo, si así lo queremos, la definición de un orden natural para los alumnos. Puedo elegir usar el apellido, el nombre, el número de matrícula, etc. De acuerdo al atributo que elija para definir el orden natural codificaré el método `compareTo()`. Lo que es importante es que la definición de este método sea compatible con el `equals()`; esto es que `a.equals(b)` si y sólo si `a.compareTo(b) == 0`.

- Una ventaja de *TreeSet* es que el orden en el que aparecen los elementos al recorrerlos es el orden natural de ellos (los objetos deberán implementar Comparable, si no lo hacen se deberá especificar una función de comparación manualmente).
- Una desventaja es que mantener todo ordenado tiene un costo, y esta clase es un poquito menos eficiente que *HashSet*.

7.2.5. Interfaz Map

Un *Map* (`Map<K,V>`) representa lo que en otros lenguajes se conoce como "diccionario" (antiguos `Dictionary` de Java) y que se suele asociar a la idea de "tabla hash" (aunque no se implemente necesariamente con esa técnica). Un *Map* es un conjunto de valores, con el detalle de que cada uno de estos valores tiene un objeto extra asociado. A los primeros se los llama "claves" o "keys", ya que nos permiten acceder a los segundos.

Un *Map* no es una *Collection* (es una *Collections*) ya que esa interfaz le queda demasiado pequeña, ya que un *Map* es bidimensional (*Collection* es unidimensional). No hay una manera trivial de expresar un Map en una simple serie de objetos que podemos recorrer. Sí podríamos recorrer una serie de objetos si cada uno de ellos representase un par {clave, valor} (y de hecho eso se puede hacer). Pero esta forma de recorrer un Map no es la forma primaria en que se usa.

- `HashMap<K,V>` es el tipo de mapeo más sencillo y probablemente el más usado. Es la clase a utilizar si queremos asociar pares de claves y valores **sin orden**, sin más. Internamente, como su nombre indica, utiliza un hash para almacenar la clave. No permite claves duplicadas, pero si **utilizar `null` como clave**.

- `Hashtable<K,V>` es una vieja conocida del JDK 1.0, que, como `Vector<E>` pasó a formar parte del framework de colecciones en Java 1.2. Esta clase es muy similar a `HashMap<K,V>`, con la excepción de que **es sincronizada** y que **no acepta utilizar el valor `null` como clave**.

- `LinkedHashMap<K,V>` es una clase introducida con el J2SE 1.4 que extiende `HashMap<K,V>` y utiliza una lista doblemente enlazada para poder recorrer los elementos de la colección en el orden en el que se añadieron. Esta clase es ligeramente más rápida que `HashMap<K,V>` a la hora de acceder a los elementos, pero es algo más lenta al añadirlos.

- `TreeMap<K,V>`, en el que los pares clave-valor se almacenan en un árbol ordenado según los valores de las claves. Como es de esperar es la clase más lenta a la hora de añadir nuevos elementos, pero también a la hora de accederlos.

De entre los métodos comunes a las cuatro clases los más utilizados son:

MÉTODO	DEFINICIÓN
`void clear()`	Elimina todos los elementos de la colección.
`boolean containsKey(Object key)`	Comprueba si la clave especificada se encuentra en la colección.
`boolean containsValue(Object value)`	Comprueba si el valor especificado se encuentra en la colección.
`V get(Object key)`	Devuelve el valor asociado a la clave especificada o `null` si no se encontró.
`boolean isEmpty()`	Comprueba si la colección está vacía.
`Set keySet()`	Devuelve un conjunto con las claves contenidas en la colección.
`V put(K key, V value)`	Añade un nuevo par clave-valor a la colección.
`V remove(Object key)`	Elimina el par clave-valor correspondiente a la clave pasada como parámetro.
`int size()`	Devuelve el número de pares clave-valor que contiene la colección.
`Collection values()`	Devuelve una colección con los valores que contiene la colección.
`Collection entrySet()`	Devuelve todos los elementos, en formato clave,valor

7.2.6. Interfaz Queue

Aparece en Java 1.5 para la creación de colas de prioridad (*PriorityQueue*), es decir, colas que están ordenadas no solo por el orden de llegada sino también por un criterio, por ejemplo, el que hemos hablado antes "orden natural" (y todo lo que ello conlleva).

Las operaciones que suelen admitir las colas son "encolar", "obtener siguiente", etc. Y por lo general siguen un patrón que en computación se conoce como FIFO (por la sigla en inglés de *"First In - First Out" - "lo que entra primero, sale primero"*), lo que no quiere decir otra cosa que lo obvio: El orden en que se van obteniendo los "siguientes" objetos coincide con el orden en que fueron introducidos en la cola.

Hasta hace poco, para implementar una cola FIFO en Java la única opción provista por la biblioteca de colecciones era *LinkedList*. Como ya se dijo más arriba, esta implementación ofrece una implementación eficiente de las operaciones "poner primero" y "sacar último".

Los métodos necesarios para usar una *LinkedList* como una cola eran parte solamente de la clase *LinkedList*, no existía ninguna interfaz que abstraiga el concepto de "cola". Esto hacía imposible crear código genérico que use indistintamente diferentes implementaciones de colas (que por ejemplo no sean FIFO sino que tengan algún mecanismo de prioridades).

Esta situación cambió recientemente a partir del agregado a Java de dos interfaces expresamente diseñadas para el manejo de colas: La interfaz *Queue* tiene las operaciones que se esperan en una cola. También se creó *Deque*, que representa a una "double-ended queue", es decir, una cola en la que los elementos pueden añadirse no solo al final, sino también "empujarse" al principio.

7.2.7. Resumen Colecciones

El siguiente cuadro nos puede ayudar a la hora de decidir implementar una interfaz u otra dependiendo el escenario que queramos llevar a cabo:

Class	Map	Set	List	Ordered	Sorted
HashMap	X			No	No
Hashtable	X			No	No
TreeMap	X			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	X			By insertion order or last access order	No
HashSet		X		No	No
TreeSet		X		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		X		By insertion order	No
ArrayList			X	By index	No
Vector			X	By index	No
LinkedList			X	By index	No
PriorityQueue			x	Sorted	By to-do order

7.3. Ordenación de Arrays y Colecciones de Objetos

Un asunto importante que tenemos que conocer es el concerniente al conocimiento de los métodos para ordenar conjuntos de objetos, así como la definición de criterios de ordenación, para este propósito, juegan un papel importante las interfaces *Comparable* y *Comparator*.

Para que un array o colección de objetos pueda ser ordenador según el "orden natural" de los objetos, la clase a la que estos pertenecen debe definir el criterio de comparación de los objetos, estableciendo así las reglas de ese "orden natural" que serán utilizadas por los métodos de ordenación.

7.3.1. Implementación de Comparable

En temas anteriores hemos visto que esta interfaz proporciona el método `compareTo()` para permitir que las clase que la implementan puedan definir los criterios de comparación de objetos. La definición de esta interfaz es:

```
interface Comparable<T>{
      int compareTo(T obj);
}
```

siendo `obj` el objeto **contra** el que se realiza la comparación.

Al fin de notificar el resultado de la comparación, el método `compareTo()` deberá devolver un número entero cuyo valor debe cumplir el siguiente criterio:

- ***< 0*** cuando el objeto con el que se compara es mayor, según el criterio especificado en el método.
- ***= 0*** si ambos objetos son iguales.
- ***> 0*** cuando el objeto con el que se compara es menor, según el criterio especificado en el método.

Por ejemplo:

```
package colec;
public class Numeros implements Comparable<Numeros>{
      private int numero;
      public Numeros(int n){this.numero = n;}

      public int getNumero(){return numero;}

      public int compareTo(Numeros num){
            return this.numero-num.getNumero();
      }
}
```

```
package colec;
public class ProbarNumeros{

	public static void main(String... args){
		Numeros num1 = new Numeros(1);
		Numeros num2 = new Numeros(2);

		// Numeros num2 = new Numeros(1);

		System.out.printf("IGULAES?=%d",num2.compareTo(num1));
	}
}
```

Salida:

NOTA_1

Es importante recordar que cuando sobrescribimos `equals()`, debemos tomar un argumento de tipo `Object`. Sin embargo, cuando sobrescribimos `compareTo()`, debemos tomar un argumento del tipo que estamos ordenando.

NOTA_2

Tambien debemos de saber que tanto `String` como las clases envoltorio implementan la interfaz `Comparable`, sin embargo, ni `StringBuffer` ni `StringBuilder` la implementan.

7.3.2. Implementación de Comparator

Para el caso de la Nota_2, si necesitamos ordenar conjuntos de objetos cuyas clases no implementan la interfaz `Comparable`, podemos recurrir a otra técnica consistente en la definición de una clase adicional, donde se establezcan los criterios de comparación. Esta nueva clase deberá implementar la interfaz `Comparator`.

La interfaz `Comparator` se encuentra en el paquete `java.util`. Proporciona un método llamado `compare()` en el que se define el criterio de comparación entre los objetos de una determinada clase, y su formato es:

```
int compare(T obj1, T obj2)
```

donde `obj1` y `obj2` son los objetos que se van a comparar y `T` la clase a la que pertenecen.

En cuanto al valor devuelto, se debe seguir el siguiente criterio:

- ***< 0*** cuando `obj1` es menor que `obj2`, según el criterio especificado en el método.
- ***= 0*** si ambos objetos son iguales.
- ***> 0*** cuando `obj1` es mayor que `obj2`, según el criterio especificado en el método.

Para probar el funcionamiento de esta interfaz vamos a redefinir la clase Numeros anterior. En este caso necesitamos tener una clase mas, nosotros la llamaremos Comparadora, donde vamos a implementar la interfaz Comparator para establecer el criterio de comparación de objetos de la clase Numeros:

```
package colec;
public class Numeros2 {
	private int numero;
	public Numeros2(int n){
		this.numero = n;
	}
	public int getNumero(){
		return numero;
	}
}
```

```
package colec;
import java.util.*;

public class Comparadora implements Comparator<Numeros2>{
	public int compare(Numeros2 num1, Numeros2 num2){
		return num1.getNumero()-num2.getNumero();
	}

	public boolean equals(Object obj){
		return true;
	}
}
```

```
package colec;
public class ProbrarNumeros2{

	public static void main(String... args){
		Numeros2 num1 = new Numeros2(1);
		Numeros2 num2 = new Numeros2(2);

		Comparadora com = new Comparadora();
		System.out.printf("IGUALES?=%d",com.compare(num1,num2));
	}
}
```

```
Salida:
```

7.4. Genéricos

Para no hacer muy extenso este tema nos vamos a centrar en la modalidad genérica de las interfaces Set, List, Queue y Map, así como las clases que implementan.

7.4.1. Los parámetros de tipo

Vamos a estudiar cómo se usaría de forma correcta los parámetros de tipo en la declaración de las clases/interfaces, variables de instancia, argumentos de método y tipos de retorno.

En el siguiente ejemplo podemos ver que se corresponde a la definición de una clase llamada Pila que se va a implementar como un tipo genérico, utilizando parámetros de tipo en la declaración de la propia clase en el los tipos de los métodos:

```
public class Pila<E>{
	private ArrayList<E> datos;

	public Pila(){
		datos = new ArrayList<E>();
	}

	public void agregar(E dato){
		datos.add(dato);
	}

	public E recuperar(int pos){
		return datos.get(pos);
	}
}
```

Ahora si queremos crear una instancia de esta clase y utilizarla para almacenar objetos de tipo String, por ejemplo, para después recuperar su contenido, tendremos que hacer:

```
Pila<String> pil = new Pila<String>();
pil.agregar("Juan Carlos");

System.out.println(pil.recuperar(0));
```

Otra situación frente la cual nos podemos encontrar, es la de tener que declarar alguna clase, interfaz o método de tipo genérico en donde tengamos que parametrizarla de tal forma que se da una situación de herencia o implementación de una interfaz, para ello tendremos que seguir la siguiente sintaxis:

```
public class Prueba <T extends Runnable&Serializable>{
…
}
```

En este caso vemos que la palabra reservada `extends` sirve tanto para indicar la herencia de clases como implementación de interfaces. Además como estamos implementando mas de una interfaz los nombres de estas deben especificarse separadas por el símbolo "&".

Además podemos vernos obligados a utilizar mas de un parámetro en la declaración de tipos, esta situación la subsanaremos:

```
public interface Prueba2 <K,V extends Runnable>{
…
}
```

7.4.2. Comodines

¿Podríamos definir métodos polimórficos que pudiesen trabajar con colecciones genéricas? Con lo visto hasta ahora no, pero para eso Java incluye los comodines, imaginemos la situación siguiente:

Queremos un método que muestre por pantalla el contenido de cualquier colección de objetos utilizando para ello los tipos genéricos, podríamos hacer lo siguiente:

```
public void imprimeContenido(ArrayList<Object> objs){
      for(Object o:objs){
            System.out.println(o.toString());
      }
}
```

Pero no podríamos llamar a este método por ejemplo con un ArrayList<String> es decir, de un tipo especificado ya que provocaría un error de compilación:

```
imprimeContenido(new ArrayList<String>);
```

Para este tipo de situaciones tenemos que hacer uso de los comodines, para solucionar este problema bastaría con redefinir el método de la siguiente manera:

```
public void imprimeContenido(ArrayList<?> objs){
	for(Object o:objs){
		System.out.println(o.toString());
	}
}
```

El símbolo ? representa el "tipo comodín", es decir, cualquier tipo de objeto. Según esta definición el parámetro `ArrayList<?>` objs en nuestro método puede recibir un ArrayList de cualquier tipo.

Algo curioso de los comodines es que se pueden usar de forma conjunta con la palabra extends para limitar el rango de los objetos admitidos a un determinado subtipo. Imaginemos que queremos un método que muestre por pantalla el sueldo de todos los Empleados almacenados en una Colección:

```
imprimeSueldo(ArrayList<?> empleados)
```

Aunque la implementación es correcta, puede resultar un poco peligroso, puesto que un *ArrayList* de objetos *String* o cualquier otra clase también serían admitidos en la definición de nuestro método, con el siguiente riesgo de provocar una excepción durante la ejecución.

La opción más segura para este tipo de situaciones es la de limitar el parámetro a todas aquellas colecciones cuyos objetos sean subclases de Empleado, de la siguiente manera:

```
public void imprimeSueldo(ArrayList<? extends Empleado> empleados){
	for(Empleado e:empleados){
		System.out.println(e.getSueldo());
	}
}
```

Algo muy importante y con lo que tenemos que tener mucho cuidado, es que al trabajar con un tipo comodín, tenemos que tener presenta que representa a cualquier tipo de dato pero que en **ningún caso podemos añadir a una colección de tipo comodín un objeto de un tipo especifico.** Por ejemplo, el siguiente código provocaría un error de compilación:

```
public void annadir(ArrayList<? extends Empleado> empleados){
	empleados.add(new Programador());//ERROR COMPILACIÓN
}
```

Algunos aspectos claves en el uso de comodines son:

- Los comodines se emplean en la declaración de variables de tipos genéricos para hacer referencia a "cualquier tipo", por ejemplo:

```
List<?> num = new ArrayList<Integer>();
List<?> cad = new ArrayList<String>();
```

- Un comodín puede ser utilizado tambien para limitar los argumentos de tipo generic a un determinado subtipo o supertipo, por ejemplo:

  - `void imprimir(List <? super Integer> obj)`: el método admite colecciones List que almacenen cualquier conjunto de objetos Integer o supertipos de esta clase.

  - `void imprimir(Wrapper <? extends Empleado> obj)`: el método admite objetos `Wrapper` que encapsulen componentes de tipo Empleado o cualquier subtipo de este.

- Las instancias de colecciones genéricas que utilizan el comodín como parámetro de tipo **no admiten operaciones de adicción de objetos**.

7.4.3. Métodos Genéricos

Además de las clases genéricas también es posible declarar métodos genéricos en el interior de las clases, ***sin necesidad de que estas estén declaradas como genéricas.*** Estos métodos se declaran especificando el parámetro del tipo delante del tipo de devolución del método.

Un ejemplo de esto podrían ser las siguientes declaraciones:

```
<T> void metodo1(List<T> lis)

<T, S extends T> void metodo2(Collection<T> col, S obj)
```

siendo T y S los parámetros de tipo. Podemos observar que cuando se utiliza mas de un parámetro de tipo y existe alguna relación entre estos, dicha relación deber ser especificada en la declaración del método, este es el caso del metodo2.

Un ejemplo práctico podría consistir en lo siguiente:

```
public class GestionEmpleado{
      <T extends Number> void calculaSueldo(Empleado<T> emp){
            System.out.println(emp.imprimeSuedo(0));
      }
}
```

La llamada a un método genérico se hace exactamente igual que a un método normal, en nuestro caso por ejemplo:

```
Empleado<Integer> emp = new Empleado<Integer>();
calcularSueldo(emp);
```

En general deberemos definir un método como genérico cuando existe una dependencia entre los parámetros del método o bien entre alguno de los parámetros y el tipo de devolución. De no existir esta dependencia, deberán utilizarse comodín en lugar de definirlo como genérico.

7.4.4. Uso de instanceof con genéricos

Ya sabemos que instanceof se utiliza para comprobar el tipo de un objeto. De cara a utilizar este operador con un tipo genérico, hay que tener en cuenta que **el único parámetro de tipo admitido es el comodín**. Así por ejemplo:

```
List<Integer> obj = new ArrayList<Integer>();
```

Podríamos utilizar instanceof por ejemplo en los siguientes casos:

BIEN	MAL
`obj instanceof List`	`obj instanceof List<Integer>`
`obj instanceof List<?>`	`obj instanceof List<? extends Number>`
`obj instanceof Set<?>`	

7.4.5. Genericos y Arrays

Podemos crear arrays de objetos que utilicen tipos genéricos, aunque hay que tener en cuenta que **único parámetro admitido es el comodín.** Según esto:

BIEN
`List<?>[] num = new ArrayList<?>[10];`
MAL
`List<Integer>[] num = new ArrayList<Integer>[10];`

8. Clases Anidadas

Las clases anidadas contienen en su interior la definición de otra clase. Son conocidas como clases anidadas (nested classes) o clases internas (inner class), sólo existen para el compilador, ya que éste las transforma en clases regulares separando la clase externa de la interna con el signo $.

8.1. Tipos de clases anidadas

Por clase anidada entendemos una clase que contiene en su interior la definición de otra clase, pudiéndose da diferentes situaciones y casos particulares. En función de cómo y dónde este definida la clase interna, distinguiremos cuatro tipos o situaciones que se pueden presentar:

a) Clases internar estándares
b) Clases internas locales a método
c) Clases anónimas
d) Clases internas estáticas

8.1.1. Clases internas estándares

Esta situación se da cuando tenemos un clase no estática definida en el interior de otra clase, como miembro de la misma. La sintaxis será:

```
class Externa{
      class Interna{
      }
}
```

Vamos a ver un ejemplo práctico de este tipo:

```
package innerc;

class Externa{
      String cadena = "Carlos";

      //Clase INTERNAL
      class Interna{
            public void muestra(){
                  System.out.println("Hola: "+cadena);
            }
      }
}
```

A la hora de instanciar este tipo de clases podemos encontrarnos en tres situaciones diferentes. La forma estándar, en la que la clase externa se instancia de forma normal, pero en la que para instanciar la clase interna, **será necesario disponer previamente de una instancia de la clase externa.** (Obsérvese el uso del new para la clase interna)

```
package innerc;
class PrincipalInternas{

	public static void main(String... args){
		Externa ex = new Externa();
		Externa.Interna in = ex.new Interna();
		in.muestra();
	}
}
```

Salida:

```
>javac innerc\PrincipalInternas.java

>java innerc.PrincipalInternas
Hola: Carlos
```

```
>dir innerc
24/08/2009  18:45 Juan Carlos               749 Externa$Interna.class
24/08/2009  18:45 Juan Carlos               338 Externa.class
24/08/2009  18:43 Juan Carlos               181 Externa.java
24/08/2009  18:45 Juan Carlos               513 PrincipalInternas.class
24/08/2009  18:43 Juan Carlos               197 PrincipalInternas.java
```

NOTA: Como se desprende del ejercicio anterior, **una clase interna de este tipo, tiene acceso al resto de miembros de la clase externa.**

La otra forma de instanciar una clase interna, sería hacerlo desde el interior de un método de la clase externa. Para ello modificamos la primera clase:

```
package innerc;
class Externa2{
	String cadena = "Carlos";
	void imprime(){
		Interna inte = new Interna();
		inte.muestra();
	}
	//Clase INTERNAL
	class Interna{
		public void muestra(){
			System.out.println("Hola: "+cadena);
		}
	}
}
```

Salida: Para compilarla y ejecutarla desde la clase principal:

```
	Externa2 ex = new Externa2();
	ex.imprime();
```

Todavía nos falta una situación que tratar con este tipo de clases, será el caso en el que desde la clase interna se quisiera hacer **referencia a una instancia de la clase externa**, para este cometido se utiliza el *this*. Así dado el siguiente código:

```
package innerc;
class Externa3{

	void muestraEx(){
		System.out.println("Hola desde Externa");
	}

	class Interna{
		public void muestraIn(){
			System.out.println("Hola desde Interna");
		}
		public void imprime(){
			this.muestraIn();
			Externa3.this.muestraEx();
		}
	}
}
```

```
package innerc;
class PrincipalInternas3{

	public static void main(String... args){
		Externa3 ex = new Externa3();
		Externa3.Interna in = ex.new Interna();

		in.imprime();
	}
}
```

Salida:

```
>javac innerc\PrincipalInternas3.java

>java innerc.PrincipalInternas3
Hola desde Interna
Hola desde Externa
```

Una clase interna no es más que un miembro de una clase, por lo tanto, para esta situación se soportaran los siguientes tipos de modificadores:

`final, abstract, strictfp, static, private, protected y public`

8.1.2. Clases internas locales a método

Esta situación se corresponde al caso de dos clases anidadas en donde la clase interna está definida en el interior de un método de la clase externa, la sintaxis será:

```
class Externa{
      void metodo(){
            class Interna{
            }
      }
}
```

Una clase interna definida en el interior de un método de otra clase, **solamente podrá ser instanciada en el interior de dicho método**, después de la declaración de la clase interna. Así pues:

```
package innerc;
class Externa4{

      void muestraEx(){
            //Clase Interna
            class Interna{
                  public void muestraIn(){
                        System.out.println("Hola desde Interna");
                  }
            }
            //Instancia, solo después de declararla
            Interna in = new Interna();
            in.muestraIn();
            System.out.println("Hola desde Externa");
      }
}
```

```
package innerc;
class PrincipalInternas4{

      public static void main(String... args){
            Externa4 ex = new Externa4();
            ex.muestraEx();
      }
}
```

Salida:

```
>javac innerc\PrincipalInternas4.java

>java innerc.PrincipalInternas4
Hola desde Interna
Hola desde Externa
```

Existe una restricción importante en el uso de este tipo de clases. Se trata del hecho de que una clase local a método no puede acceder a las variables locales definidas en dicho método, salvo que estén definidas como *final*. Por ejemplo, lo siguiente no compilaría:

```
package innerc;
class Externa44{
	String nombre = "Carlos";
	void muestraEx(){
		int edad = 25;
		//Clase Interna
		class Interna{
			public void muestraIn(){
				System.out.println("Hola: "+nombre);
				edad++;
			}
		}
		//Instancia, solo después de declararla
		Interna in = new Interna();
		in.muestraIn();
		System.out.println("Hola desde Externa");
	}
}
```

```
Salida:

>javac innerc\PrincipalInternas44.java

>javac innerc\PrincipalInternas44.java
.\innerc\Externa44.java:10: local variable edad is accessed from
within inner class; needs to be declared final
				edad++;
```

Dado que una clase definida dentro de un método es un elemento local a este, únicamente podrán utilizarse en su definición los modificadores de acceso:

```
final, abstract
```

8.1.3. Clases Anónimas

Es el caso mas extraño que se pueda dar dentro de las clases internas. Una clase anónima es una clase sin nombre, definida en la misma línea de código donde se crea el objeto de la clase. Esta operación se lleva a cabo en el interior de un método de otra clase, por ello la clase anónima es considerada como una clase interna anidada.

Una clase anónima siempre debe ser una subclase de otra clase existente o bien deben implementa alguna interfaz. La sintaxis para su definición es:

```
Superclase var = new Superclase(){
      //Definicion de la clase anonima
}
```

donde *var* será la variable que almacene la instancia de la clase anónima, que será una subclase de *Superclase*. Recordar que la definición de la clase anónima y la creación de una instancia de la misma representan acciones inseparables que se realizan en la misma línea de código.

```
package innerc;
class Operaciones{
      public void imprimir(){
            System.out.println("Imprimo desde Operaciones");
      }
}
```

```
package innerc;
class ExternaAnonima1{
      //Clase que tiene que existir
      Operaciones ope = new Operaciones(){
            public void imprimir(){
                  System.out.println("Imprimo desde Anonima");
            }
      };//No olvidarse del ;

      public void muestraEx(){
            ope.imprimir();
      }
}
```

```
package innerc;
class PrincipalAnonimas1{
      public static void main(String... args){
            ExternaAnonima1 ext = new ExternaAnonima1();
            ext.muestraEx();
      }
}
```

Salida:

```
>javac innerc\PrincipalAnonimas1.java

>java innerc.PrincipalAnonimas1
Imprimo desde Anonima
```

8.1.4. Clases Internas Estáticas

El último caso especial de clases anidadas es aquel en el que la clase interna resulta ser un dato miembro estático. Para construirlas la sintaxis es:

```
class Externa{
      static class Interna{
            //código clase interna
      }
}
```

Como sucede con los métodos, **las clases internas definidas como estáticas no pueden acceder a las variables y métodos no estáticos de la clase externa**.

Para la instanciación de una clase de este tipo, haríamos:

```
Externa.Interna var = new Externa.Interna();
```

Un ejemplo de esto, podría ser lo siguiente:

```
package innerc;
class ExternaEstatica{

      static class Interna{
            public void muestra(){
                  System.out.println("Imprimo desde Estatica");
            }
      }
}
```

```
package innerc;
class PrincipalEstatica{

      public static void main(String... args){
            ExternaEstatica.Interna ext = new ExternaEstatica.Interna();
            ext.muestra();
      }
}
```

Salida:

```
>javac innerc\PrincipalEstatica.java

>java innerc.PrincipalEstatica
Imprimo desde Estatica
```

9. Excepciones

En algunas de las practicas que hemos ido desarrollando hemos tenido algún "suave" contacto con las excepciones, valga como ejemplo el método *readLine()* de la clase *BufferReader* para la lectura de cadena de caracteres por teclado. En éste tuvimos que declarar la excepción *IOException* en el método *main()* para poder compilar el programa.

Otro caso práctico de las excepciones lo hemos tenido en nuestro "amigo" *NullPointerException*, que tan de vez en cuando nos ha ido saliendo cuando empezábamos a programar con Java.

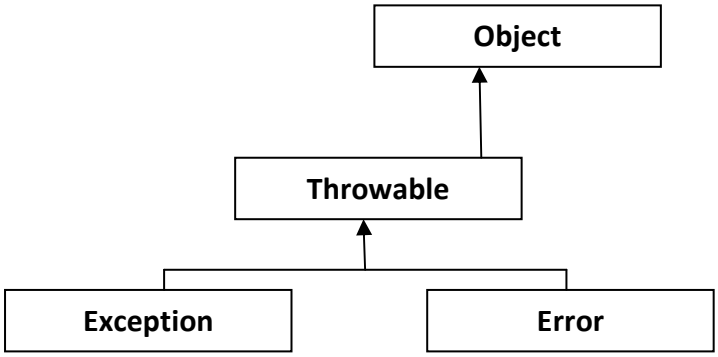
9.1. Excepciones y errores

En Java tenemos que diferenciar muy bien lo que podemos considerar una excepción y lo que es un error. **Una excepción** es una situación anómala que puede producirse durante la ejecución de un programa, como puede ser la división por cero, un acceso a una posición de un array que no existe, etc.

Mediante la captura de excepciones, Java proporciona un mecanismo que permite al programa sobreponerse a estas situaciones, pudiendo el programador decidir las acciones a realizar para cada tipo de excepción que pueda ocurrir.

Además de excepciones, en un programa Java también puede producirse errores. **Un error** representa una situación anormal irreversible, como por ejemplo un fallo de la maquina virtual. Por norma general, un programa no deberá intentar de un error, dado que son situaciones que se escapan al control del programador.

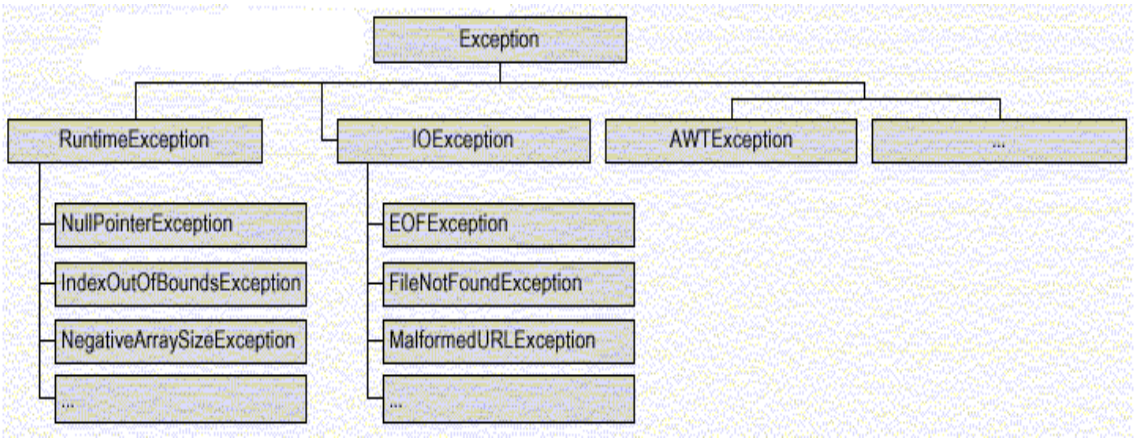
Cada tipo de excepción está representada por una subclase *Exception*, mientras que los errores se encuentran en la clase *Error*:



9.2. Clases de excepción

Al producirse una excepción se crea un objeto de la subclase Exception a la que pertenece la excepción. Como veremos mas adelante, este objeto va a ser utilizado para obtener toda la información de la misma.

En la siguiente imagen podemos ver una jerarquía de clases con algunas de las excepciones más habituales que nos podemos encontrar dentro de un programa Java:



9.3. Tipos de excepción

Desde el punto de vista del tratamiento de una excepción dentro de un programa, hay que tener en cuenta que todas estas clases de excepción se dividen en dos grandes grupos:

- Excepciones marcadas
- Excepciones no marcadas

9.3.1. Excepciones marcadas

Son aquellas cuya *captura es obligatoria*. Normalmente, este tipo de excepción se produce al invocar a ciertos métodos de determinadas clases y son generadas (lanzadas) desde el interior de dichos métodos como consecuencia de algún fallo durante la ejecución de los mismos.

Todas las clases de excepciones, salvo *RuntimeException* y sus subclases pertenecen a este tipo. Un ejemplo ya mencionado lo tenemos en el método *readLine()* de la clase *BufferReader*.

Si un bloque de código invoca a algún método que puede provocar una excepción marcada y esta no se captura, **el programa no compilará.**

Declaración de una excepción

Los métodos que pueden provocar excepciones marcadas deben declarar estas en la definición del método. Para declarar una excepción se utiliza la palabra reservada *throws*, seguida de la lista de excepciones que el método puede provocar, por ejemplo:

```
public String readLine() throws IOException
public void service(…) throws ServletException, IOException
```

Así siempre que vayamos a utilizar algún método que tenga declaradas excepciones, hemos de tener presente que estamos obligados a capturar dichas excepciones.

9.3.2. Excepciones no marcadas

Pertenecen a este tipo todas las excepciones de tiempo de ejecución, es decir, RuntimeException y todas sus subclases.

No es obligatorio capturar dentro de un programa Java una excepción no marcada, el motivo es que gran parte de ellas (NullPointerException, ClassCastException, …) se producen como consecuencia de una mala programación, por lo que una solución no debe de pasar por preparar al programa para esto. Tan solo es lógico capturar las excepciones de este tipo ArithmeticException.

Si durante la ejecución de un programa se produce una excepción y no es capturada, la Maquina Virtual provoca la finalización inmediata del mismo, enviando a la consola el volcado de la pila con los datos de la excepción a la consola. Estos datos permiten al programador detectar fallos de programación durante la depuración del mismo.

Un ejemplo de esto podría ser:

```
package excepciones;

public class ExcepcionDivision{

	public static void main(String... args){
		//Aquí se produce la excepción
		int k = 4/0;
	}
}
```

Salida:

```
>javac excepciones\ExcepcionDivision.java

>java excepciones.ExcepcionDivision
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at excepciones.ExcepcionDivision.main(ExcepcionDivision.java:6)
```

9.4. Captura de excepciones

En el momento en el que se produce una excepción en un programa, se crear un objeto de la clase de excepción correspondiente y se "lanza" a la línea de código donde la excepción tuvo lugar.

El mecanismo de captura permite atrapar el objeto de excepción lanzado por la instrucción e indicar las diferentes acciones a realizar según la clase de excepción producida.

A diferencia de las excepciones, los errores representan fallos de sistema de los cuales el programa no se puede recuperar. Esto implica que no es obligatorio tratar un error en una aplicación Java, de hecho, aunque se pueden capturar al igual que las excepciones con los mecanismos que vamos a ver a continuación, lo recomendable es no hacerlo.

9.4.1. Los bloques try…catch..finally

Estas instrucciones proporcionan una forma elegante y estructurada de capturar las excepciones dentro de un programa Java, evitando la utilización de las instrucciones de control que dificultarían la lectura del código y no harían mas que provocar mas errores.

La sintaxis para utilizar estas instrucciones es:

```
try
{
      //Instrucciones que pueden provocar excepciones
}
catch (TipoExcepcion e1)
{
      //Tratamiento Excepcion1
}
catch (TipoExcepcion e2)
{
      //Tratamiento Excepcion2
}
finally
{
      //Ultimas instrucciones a ejecutar
}
```

try

Delimita aquella o aquellas instrucciones donde se puede producir una excepción. Cuando esto sucede, el control del programa se transfiere al bloque *catch* definido para el tipo de excepción que se ha producido, pasándole como parámetro la excepción lanzada. Opcionalmente, se puede disponer de un bloque *finally* en el que definir un grupo de instrucciones de obligada ejecución.

catch

Define las instrucciones que deberán de ejecutarse en caso de que se produzca un determinado tipo de excepción. Sobre la utilización de los bloques catch, debemos tener en cuenta:

- Se pueden definir tantos bloques catch como se considere necesario. Cada bloque catch servirá para tratar un determinado tipo de excepción, **no pudiendo haber dos o mas catch que tengan declarada la misma clase de excepción.**

- Un bloque catch sirve para capturar cualquier tipo de excepción que se corresponda con el tipo declarado o cualquiera de sus subclases, por ejemplo:

```
catch (RuntimeException ecp){
      //Ultimas instrucciones a ejecutar
}
```

  se ejecutaría al producirse cualquier excepción de tipo *NullPointerException*, *ArithmeticException*, etc.

- Aunque haya varios tipos posibles de catch que puedan capturar una excepción, **solo uno de ellos será ejecutado cuando esta se produzca**. La búsqueda de los bloques catch se realiza de forma secuencial, de modo que el primer catch coincidente será el que se ejecutara.

```
package excepciones;
public class ExcepcionDivision2{

      public static void main(String... args){
            try{
                  int k = 4/0;
                  System.out.println("Sigue sin error-----");
            }catch(ArithmeticException e1){
                  System.out.println("ArithmeticException-----");
            }catch (Exception e2){
                  System.out.println("Exception-----");
            }
            System.out.println("Final del main-----");
      }
}
```

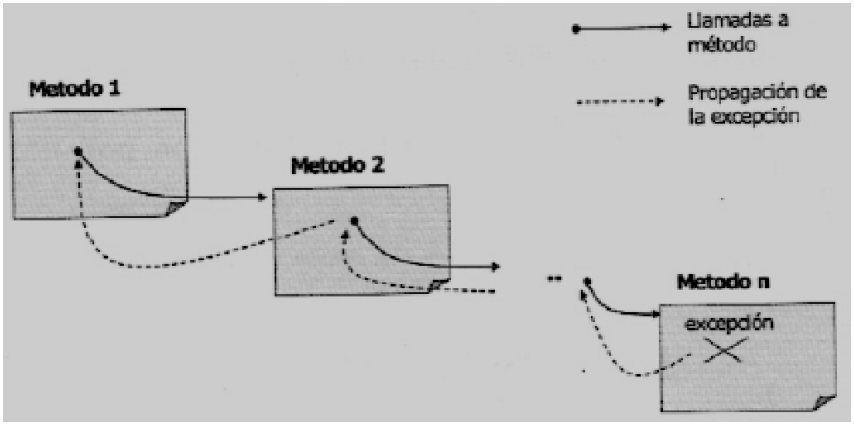
- Del punto anterior deducimos otra cosa: tras la ejecución de un bloque catch, **el control del programa nunca se devuelve al lugar donde se ha producido la excepción.**

- En el caso de que existan varios *catch* cuyas excepciones están relacionados por la herencia, los catch más específicos deben estar situados por delante de los más genéricos. De no ser así se produciría un **error de compilación** puesto que los bloques mas específicos nunca se ejecutarán.

COMPILA	NO COMPILA

try{
…
}
catch (IOExcepcion e1){
…
}
catch (Excepcion e2){
…
}
	try{
	…
	}
	catch (Excepcion e1){
	…
	}
	catch (IOExcepcion e2){
	…
	}

- Si se produce una excepción no marcada para la que no se ha definido un bloque *catch*, esta será propagada por la pila de llamadas hasta encontrar algún punto en que se trata la excepción, sino lo encuentra la JVM, abortará el programa y enviara un volcado de pila a la consola:



- Los bloques catch son opcionales. Siempre que exista un bloque *finally*, la creación de los bloques *catch* después de un *try* es opcional. Si no se encuentra el bloque *finally*, entonces es obligatorio disponer de, al menos, un bloque *catch*.

finally

Su uso es opcional y se ejecutara tanto si se produce una excepción como si no, garantizando así que un determinado conjunto de instrucciones siempre sean ejecutables.

Si se produce una excepción en *try*:

- El bloque *finally* se ejecutara después del catch para tratamiento de la excepción.
- En caso de que no hubiese ningún catch para el tratamiento de la excepción producida, el bloque *finally* se ejecutaría antes de propagar la excepción.
- Si no se produce excepción alguna en el interior de *try*, el bloque *finally* se ejecutara tras la última instrucción del *try*.

Un ejemplo de esto sería:

```
package excepciones;
public class ExcepcionDivision3{

	public static void main(String... args){
		try{
			int k = 4/0;
			System.out.println("Sigue sin error-----");
		}catch(ArithmeticException e1){
			System.out.println("Division por 0-----");
			return;
		}finally{
			System.out.println("Finally-----");
		}
		System.out.println("Final del main-----");
	}
}
```

Salida:

```
>javac excepciones\ExcepcionDivision3.java

>java excepciones.ExcepcionDivision3
Division por 0-----
Finally-----
```

NOTA: Si nos fijamos en el código y posteriormente en la salida que provoca la ejecución del mismo, podemos observar que aunque tengamos un *return* dentro del código, el bloque *finally* se ejecutara antes de que este suceda.

9.4.2. Propagación de una excepción

Ya hemos dicho que si en el interior de un método se produce una excepción que no es capturada, bien porque no está dentro de un try o bien porque no existe un catch para su tratamiento, esta se propagara en una pila de llamadas.

En el caso de las excepciones marcadas, hemos visto como estas deben de ser capturadas obligatoriamente en un programa. Sin embargo, en el caso de que no se tenga ninguna acción particular para el tratamiento de una determinada excepción de este tipo, es posible propagar la excepción sin necesidad de capturarla, dejando que sean otras partes del programa las encargadas de definir las acciones para su tratamiento.

Para propagar una excepción sin capturarla, basta con declararla en la cabecera del método en cuyo interior puede producirse, por ejemplo:

```
package excepciones;
import java.io.*;

public class Excepciones1{

	static  void  imprimeCadena(BufferedReader  bf)throws  IOException{

		//Puede provocar una excepción
		String cad = bf.readLine();
		System.out.println(cad);
	}

	public static void main(String... args){
		BufferedReader buf =
			new  BufferedReader(new  InputStreamReader(System.in));

		try{
			imprimeCadena(buf);
		}catch(IOException e1){
			System.out.println("Fallo en la lectura");
		}
	}
}
```

En el ejemplo anterior, es el método *main()* el que se encarga de recoger la excepción que se producirá en el método *imprimeCadena().* Si hubiéramos optado por propagar también en el *main()*, al ser el ultimo método de la pila de llamadas esta se propagara a la JVM y esta interrumpirá la ejecución del programa y generar un volcado de pila en la consola.

9.4.3. Lanzamiento de una excepción

En determinados casos puede resultar útil generar (crear) y lanzar una excepción desde el interior de un determinado método. Esto puede utilizarse como un medio para enviar un aviso a otra parte del programa, indicándole que algo está sucediendo y no es posible continuar con la ejecución normal del método.

Para lanzar una excepción desde el código utilizaremos la expresión:

```
throw objeto_excepcion;
```

donde `objeto_excepcion` es un objeto de alguna subclase de Exception, por ejemplo:

```
package excepciones;
public class ExcepcionCuenta{

	private double saldo;

	public ExcepcionCuenta(){
		this.saldo = 0;
	}

	public void ingresarDinero(double saldo){
		this.saldo = this.saldo + saldo;
	}
	public void setSaldo(double saldo){this.saldo = saldo;}
	public double getSaldo(){return this.saldo;}

	//método marcado para provocar excepción
	public void sacarDinero(double money) throws Exception{
		if(getSaldo() < money){
			//Creamos y lanzamos la excepción
			throw new Exception();
		}
		else{this.saldo -= money;}
	}
}
```

```
package excepciones;
public class ExcepcionCajero{

	public static void main(String... args){
		ExcepcionCuenta cue = new ExcepcionCuenta();
		try{
			cue.ingresarDinero(100);//cue.ingresarDinero(100);
			cue.sacarDinero(20);   //cue.ingresarDinero(1000);

			System.out.println("Nos quedan: "+cue.getSaldo());
		}
		catch(Exception e){
			System.out.println("No hay saldo suficiente");
		}
	}
}
```

```
Salida:
```

9.4.4. Métodos para el control de una excepción

Todas las clases de excepción heredan una serie de métodos de la clase *Throwable* que pueden ser utilizados en el interior de los catch para completar las acciones de tratamiento de una excepción.

Los métodos más importantes son:

- `String` **`getMessage()`**: devuelve un mensaje de texto asociado a la excepción, dependiendo del tipo de objeto de excepción sobre el que se aplique.

- `void` **`printStackTrace()`**: envía a la consola el volcado de la pila asociado a la excepción. Su uso puede ser tremendamente útil durante la fase de desarrollo de la aplicacion, ayudando a detectar errores de programación causantes de muchas excepciones.

- `void` **`printStackTrace(PrintStream s)`**: nueva versión del método anterior en la que se envía a la consola el volcado de la pila a un objeto *PrintStream* cualquiera, como por ejemplo, un fichero log.

```
package excepciones;
public class ExcepcionDivision3{

	public static void main(String... args){
		try{
			int k = 4/0;
			System.out.println("Sigue sin error-----");
		}catch(ArithmeticException e1){
			System.out.println("Division por 0-----");
			System.out.println("Mensaje: "+e1.getMessage());
			e1.printStackTrace();
			return;
		}finally{
			System.out.println("Finally-----");
		}
		System.out.println("Final del main-----");
	}
}
```

```
>java excepciones.ExcepcionDivision3
Division por 0-----java.lang.ArithmeticException: / by zero
Mensaje: / by zero
Finally-----

	at excepciones.ExcepcionDivision3.main(ExcepcionDivision3.java

>javac excepciones\ExcepcionDivision3.java

>java excepciones.ExcepcionDivision3
Division por 0-----
Finally-----
```

9.5. Clases de excepción personalizadas

En el caso de que un método necesite lanzar una excepción como forma de notificar una situación anómala, puede suceder que las clases de excepción existentes no se adecuen a las características de la situación que se quiere notificar.

Un ejemplo claro está en nuestra notificación del ejercicio de la cuenta bancaria, en ese caso, no tiene sentido lanzar un *NullPointerException* o *IOException* cuando se produce una situación de saldo insuficiente.

En estos casos resulta mas practico definir una clase de excepción personalizada, que será una subclase de *Exception*, que se ajuste mas a las características de la excepción que se va a tratar.

Refinando el ejercicio anterior, obtendríamos:

```
package excepciones;
public class ExcepcionCuenta2{

	private double saldo;

	public ExcepcionCuenta2(){
		this.saldo = 0;
	}

	public void ingresarDinero(double saldo){
		this.saldo = this.saldo + saldo;
	}
	public void setSaldo(double saldo){this.saldo = saldo;}
	public double getSaldo(){return this.saldo;}

	//método marcado para provocar excepción
	public void sacarDinero(double money)
					throws SaldoInsuficienteException{
		if(getSaldo() < money){
			//Creamos y lanzamos la excepción
			throw new SaldoInsuficienteException("Sin dinero");
		}
		else{
			this.saldo -= money;
		}
	}
}
```

```
package excepciones;
public class ExcepcionCajero2{

	public static void main(String... args){
		ExcepcionCuenta2 cue = new ExcepcionCuenta2();
		try{
			cue.ingresarDinero(100);
			cue.sacarDinero(200);

			System.out.println("Nos quedan: "+cue.getSaldo());
		}
		catch(SaldoInsuficienteException e){
			System.out.println(e.getMessage());
		}
	}
}
```

```
package excepciones;
public class SaldoInsuficienteException extends Exception{
	public SaldoInsuficienteException(String mensaje){
		super(mensaje);
	}
}
```

Salida:

```
>javac excepciones\ExcepcionCajero2.java excepciones\ExcepcionCuenta2.

>java excepciones.ExcepcionCajero2
Sin dinero
```

La cadena de texto recibida por el constructor permite personalizar el mensaje de error obtenido al llamar al método `getMessage()`, para ello, es necesario suministrar dicha cadena al constructor de la clase `Exception`.

PRÁCTICA 7 - Excepciones 1

PRÁCTICA 7 - Excepciones 2

PRÁCTICA 7 - Excepciones 3

9.6. Aserciones

Tenemos disponibilidad de las aserciones desde la versión Java 1.4, se utilizan durante la fase de desarrollo y depuración de una aplicación para verificar ciertas suposiciones asumidas por el programa, evitando la utilización innecesaria de instrucciones *println()* o de captura de excepciones.

Cuando la suposición asumida por el programa no se cumple, la aserción generará un error que provocará la interrupción inmediata del programa. El aspecto más positivo de las aserciones está en que **solamente se pueden habilitar durante la fase de desarrollo y depuración.** Al realizar el despliegue de la misma todas las aserciones serán ignoradas sin necesidad de introducir cambios en el código, dejando cualquier tipo de sobrecarga que estas instrucciones pudieran producir.

9.6.1. Formato de una aserción

La sintaxis para éstas es:

```
assert(condicion);
```

donde `condicion` es una expresión cuyo resultado debe de ser de tipo *boolean*. La condición siempre se espera que sea verdadera (*true*), si es así, no pasa nada, y el programa continuara ejecutándose normalmente, pero si la condición es falsa (*false*), el programa se interrumpirá lanzando un *AssertError* que **no deberá ser capturado.**

En el siguiente ejemplo, tenemos un método que vamos a utilizar para calcular la suma de unos números naturales (enteros positivos) por lo tanto podemos indicar que los parámetros a sumar tienen que ser positivos:

```
public class Aserciones1{

	private void sumaNumeros(int num1, int num2){
		assert(num1 > 0);
	}

	public static void main(String... args){
		Aserciones1 ase = new Aserciones1();
		ase.sumaNumeros(-12,12);
	}
}
```

Si el numero(s) suministrado(s) al método no fuera positivo (en este caso solo comprobamos uno), se lanzaría un error avisándonos de que algo no va como esperábamos.

Existe una segunda forma de utilizar aserciones que permite además enviar información adicional a la consola cuando se produce una aserción, la sintaxis seria:

```
assert(condicion):expresion;
```

```
public class Aserciones1{

      private void sumaNumeros(int num1, int num2){
            assert(num1 > 0): num1+" no positivo";
      }

      public static void main(String... args){
            Aserciones1 ase = new Aserciones1();
            ase.sumaNumeros(-12,12);
      }
}
```

Salida:

```
Exception in thread "main" java.lang.AssertionError: -12 numero no positivo
      at Aserciones1.sumaNumeros(Aserciones1.java:5)
      at Aserciones1.main(Aserciones1.java:10)
```

9.6.2. Habilitar aserciones

De forma predeterminada, las aserciones están inhabilitadas. Si se quiere hacer uso de ellas, primeramente habrá que indicar al compilador que compile con aserciones, para después habilitarlas en el momento de la ejecución.

Compilar con aserciones

Para compilar con aserciones la clase Aserciones1.java, deberemos de utilizar la opción –source del compilador javac.exe, tal y como se indica a continuación:

```
javac –source 1.X Aserciones1.java

      (siendo X la versión de Java que estemos utilizando)
```

Ejecutar con aserciones

Para habilitar las aserciones en tiempo de ejecución, utilizaremos:

```
java –ea Aserciones1 ó java –eneableassertions Aserciones1
```

En el caso de disponer de paquetes:

```
      java –ea:mipaquete… mipaquete.Aserciones1
(los … indica que afecta al paquete especificado y sus subpaquetes)

      java –ea:aserciones aserciones.Aserciones1
```

Uso apropiado de las aserciones

Ya que son un mecanismo de ayuda para la depuración de programas, es necesario conocer ciertas normas a tener en cuenta a la hora de utilizarlas de cara a hacer un uso apropiado de las mismas:

- No se debe utilizar para validar argumentos de un método público. De que no tenemos control sobre los posibles valores que se puedan pasar al método, no tiene sentido realizar suposiciones sobre los mismos.

- Por el mismo motivo que en el punto anterior, tampoco se deben utilizar las aserciones para validar argumentos de la línea de comandos.

- Se pueden utilizar para validar argumentos de un método privado, ya que disponemos de un control total sobre los argumentos del mismo, y es lógico hacer ciertas suposiciones sobre estos.

- Se pueden utilizar en métodos públicos, para validar casos que se sabe que nunca sucederán. Por ejemplo, puede ser útil para lanzar un error en el caso de que el programa alcance una línea de código que se supone que nunca se va a ejecutar:

```
switch(x):
      case1: System.out.println("HOLA");break;
      case2: System.out.println("ADIOS");break;
      …
      default: assert(false);
```

- No se deben utilizar cuando puedan causar efectos colaterales. Una aserción siempre debe dejar al programa en el mismo estado en el que se encontraba antes de ejecutarse ésta:

```
public class Aserciones2{
      int a;
      public void metodo(){
            assert(cambia());
      }

      public boolean cambia(){
            a++;
            return true;
      }
      public static void main(String... args){
            Aserciones2 ase = new Aserciones2();
            ase.metodo();
      }
}
```

PRÁCTICAS

PRÁCTICA 5.6.3 – Entrada de Datos

Vamos a escribir un programa que juegue con el usuario a adivinar un número. El ordenador debe generar un número aleatorio entre 1 y 100 y el usuario tiene que intentar adivinarlo.

Para ello, cada vez que el usuario introduce un valor el ordenador debe de decirle al usuario si el número que tiene que adivinar es mayor o menor que el que ha introducido. Cuando consiga adivinarlo debe indicárselo e imprimir en pantalla el número de veces que el usuario ha intentado adivinar el número. Si el usuario introduce algo que no es un número debe indicarlo de esta forma en pantalla y contarlo como un intento.

PRÁCTICA 6.3 – Encapsulación

Se trata de desarrollar un programa para la gestión de una lista de nombres. Para ello implementaremos una especie de agenda que almacene el nombre, teléfono y DNI de las personas que queremos registrar.

Las opciones que se presentarán al iniciar el programa son:

1. Agregar persona
2. Buscar persona
3. Eliminar persona
4. Mostrar todas las personas
5. Salir

Cuando se elija la opción 1, el programa solicitara el DNI, nombre y el teléfono de la persona, añadiendo dichos datos a la lista (de tamaño limitado). Tenemos que tener en cuenta que no puede haber dos personas con el mismo DNI, por lo que si se da esta circunstancia tendremos que avisar al usuario de este hecho.

La opción 2 solicitara el DNI de la persona que se quiere localizar, si se encuentra se mostraran sus datos, si no, se indicara esta circunstancia al usuario. La opción 3 es prácticamente igual que la 2 ya que la eliminación de una persona se hará a través de su DNI.

Finalmente la opción 4 mostrara los datos (DNI, nombre y teléfono) de todas las personas registradas.

Para llevar a cabo la práctica, deberíamos de disponer de una clase de encapsulación, llamada Persona, donde se guarden los tres datos básicos que se solicitan. Además se debería de encapsular toda la lógica de gestión de la agenda en una clase Agenda, por ejemplo con los siguientes métodos:

- `boolean agregar(String dni, String nombre, long telefono)` : añade a la agenda la persona con los datos indicados, devolviendo true si se ha podido añadir y false en caso contrario.
- `boolean eliminar(String dni)`: elimina a la persona con el DNI solicitado, devolviendo true en caso de éxito y false en caso contrario.
- `Persona recuperar(String dni)`: devuelve a la persona con el DNI especificado, si no existe devuelve null.
- `Enumeration total()`: devuelve una enumeración con todos los DNI.

Sería lógico que además partiéramos de una clase Principal, encargada de efectuar las operaciones de entrada/salida.

PRÁCTICA 6.5 - Constructores

Se pretende desarrollar una aplicación que simule el funcionamiento de un cajero automático. Para ello primero creamos una clase llamada Cuenta, que se va a encargar de gestionar las operaciones sobre la cuenta. Además de los constructores y campos que se estimen necesarios, la clase contendrá los métodos:

- `void ingresar(float c)`: agregue saldo a la cuenta recibida.
- `void extraer(float c):` descuenta saldo a la cuenta recibida. Tras la llamada a este método , el saldo podrá quedar negativo.
- `float getSaldo():` devuelve el saldo actual.

Por otro lado existirá una clase con el método *main* encargada de la captura y presentación de los datos y la gestión de la cuenta. Al iniciarse la aplicación se mostrara el siguiente menú:

1) Crear cuenta vacía.
2) Crear cuenta saldo inicial.
3) Ingresar dinero.
4) Sacar dinero.
5) Ver saldo.
6) Salir.

Donde la opción 1 crea un objeto con saldo 0, la opción 2 solicita una cantidad y crea un objeto Cuenta con ese saldo inicial. En la opción 3 se solicita una cantidad y la ingresa en el objeto creado por la opción 1 o 2, mientras que en la 4 se solicita una cantidad y la extrae de objeto creado en la opción 1 o 2.

Finalmente la opción 5 muestre el saldo de todas las cuentas activas, y la opción 6 provoca que el/los objetos creados se destruyen y se pierda los saldos de todos.

Mientras no se elija la opción 6 o la opción sea incorrecta el menú se vuelve a presentar continuamente.

PRÁCTICA 6.6.1 – Herencia

Vamos a poner en práctica los conceptos de la herencia, para ello vamos a desarrollar una clase que herede de *BufferedReader* y que además de tener los métodos que ésta posee, incluya una seria de métodos propios. Para ello la nueva clase, a la que llamaremos *LecturaNumeros*, deberá de definir los siguientes métodos adicionales:

- `int readInt()`: devolverá el dato numérico correspondiente a la última línea de caracteres suministrados.
- `int readInt(String mensaje)`: igual que el anterior, mostrando previamente al usuario el mensaje indicado.
- `Integer readInteger()`: funciona igual que `readInt()`, devolviendo el dato para obtener el objeto.
- `double readDouble()`: devolverá el dato numérico leído, como un tipo doublé.
- `double readDouble(String mensaje)`: igual que el anterior, mostrándole al usuario el mensaje indicado.

Así mismo la clase deberá de contar con una seria de constructores que permitan asociar el objeto al dispositivo de entrada:

- `LecturaNumeros()`: permite al objeto para realizar la lectura de datos por teclado.
- `LecturaNumeros(Reader r)`: realiza la lectura desde el objeto *Reader* especificado como parámetro.

Después de la creación de esta clase, realizaremos una clase de prueba en la que se utilice *LecturaNumeros* para solicitar cinco números por teclado al usuario y mostrar el resultado en pantalla.

PRÁCTICA 6.6.2 - Sobrescritura

Desarrollar una nueva versión de la PRÁCTICA 6.5, en la que se utilizara una nueva clase que gestione las operaciones sobre la cuenta, llamada *CuentaClave*. Esta clase será una subclase de Cuenta y tendrá las siguientes características:

- Incluirá un nuevo dato miembro llamado *clave.*
- Sobrescribirá el método *extraer()* de modo que solo permita la extracción si hay saldo suficiente, sino no hará nada.

En cuanto al funcionamiento del programa será igual que en el caso anterior, solo que al elegir las opciones 1 y 2 para la creación de la cuenta, se pedirá también al usuario la clave que se le permite asociar, aunque luego no se utilice en las restantes operaciones.

No se enviará ningún tipo de aviso al usuario en el caso de que intente sacar más dinero del que dispone.

PRÁCTICA 6.8 – Polimorfismo

Utilizando la clase Figura, Triangulo, Rectangulo y Circulo analizadas en este punto y haciendo uso del polimorfismo, desarrollar una aplicación para la realización de cálculo de figuras.

Al comenzar el programa mostrara el siguiente menú:

a) Crear Triangulo
b) Crear Circulo
c) Crear Rectangulo
d) Salir

Cuando elija una de las tres primeras opciones, se solicitara al usuario la introducción de los atributos correspondientes, incluido el color. Tras ello, el programa mostrara el área de la figura correspondiente y su color, volviendo a aparecer el menú anterior hasta que el usuario elija la opción de salir.

PRÁCTICA 6.9 – Interfaces 1

Vamos a escribir una aplicación para una biblioteca que va a contener libros y revistas. Se van a seguir las siguientes indicaciones:

1) Las características comunes que se almacenan tanto para las revistas como para los libros son el código, el titulo y el año de publicación. Estas tres características se pasan por parámetro en el momento de crear los objetos.
2) Los libros tienen además un atributo prestado. Los libros cuando se crean no están prestados.
3) Las revistas tienen un número. En el momento de crear las revistas se pasa por parámetro este número.
4) Tanto las revistas como los libros deben tener (aparte de los constructores) un método `toString()` que devuelve el valor de todos los atributos en una cadena de caracteres. También tiene un método que devuelve el año de publicación y otro para el código.
5) Para prevenir posibles cambios en el programa se tiene que implementar una interfaz `Prestable` con los métodos:
   a. `void prestar()`
   b. `void devolver()`
   c. `boolean prestado()`
   d. La clase `Libro` implementa esta interfaz.

PRÁCTICA 7 – Excepciones 1

Se trata de realizar una nueva versión de la PRÁCTICA 6.5. La funcionalidad de la aplicación será la misma, introduciendo el mecanismo de excepciones para tratar la situación de saldo insuficiente,

Por un lado, habrá que modificar la clase Cuenta para que en caso de que se intente extraer una cantidad de dinero superior al saldo, se impida realizar dicha operación y se lance una excepción personalizada de tipo "SaldoInsuficienteException".

Por otro lado, esta excepción deberá ser capturada desde el programa principal.

PRÁCTICA 7 – Excepciones 2

Vamos a escribir una clase, de nombre Atleta, y vamos a tener en cuenta:

1) La clase tendrá un atributo entero de nombre energía.
2) La clase tendrá un método constructor que reciba por parámetro una cantidad de energía que asignará al atributo anterior.
3) La clase tendrá un método, de nombre `recargarEnergia`, que recibirá por parámetro una cantidad de energía que será sumada al atributo energía.
4) La clase tendrá un método, de nombre `correr`, que mostrara por pantalla un mensaje y decrementara la energía en 10 unidades. Antes de proceder al decremento comprobara que la energía del corredor es igual o superior a 10. Si no es así, el método lanzara una excepción del tipo AgotadoExcepcion.

PRÁCTICA 7 – Excepciones 3

Escribir una clase llamada Entrenamiento, en cuyo método `main()` se creará un objeto Atleta con una energía de 50 unidades. Se hace que el corredor corra hasta que se agote 3 veces. La primera vez que se agote su energía se recargara con 30 unidades. La segunda vez que se agote se recargara con 10 unidades, y cuando se agote por tercera vez se dará el entrenamiento por concluido.

GLOSARIO

COMPILAR

Vamos al directorio donde está nuestro proyecto. Ahí creamos y hacemos nuestras fuentes .java. Es importante tener en cuenta lo siguiente:

- Los ficheros .java que dentro no ponen `package`, deben colocarse en el directorio de nuestro proyecto. El siguiente fuente debería ir en:

```
c:\path_mi_proyecto\fuente1.java

      public class fuente1 {
            ...
      }
```

- Si un fichero .java lleva dentro `package paquete;`, debemos crear en nuestro proyecto un subdirectorio paquete y meter dentro el fichero .java. El siguiente fuente debería ir en:

```
c:\path_mi_proyecto\paquete\fuente2.java

      package paquete;
      public class fuente2 {
            ...
      }
```

- Si un fichero .java lleva dentro `package paquete.subpaquete;`, debemos crear en nuestro proyecto un subdirectorio paquete y dentro de este otro subdirectorio subpaquetes y dentro de este nuestro fuente .java. El siguiente fuente debería ir en:

```
c:\path_mi_proyecto\paquete\subpaquete\fuente3.java

      package paquete.subpaquete;
      public class fuente3 {
      ...
      }
```

Es decir, los `package` y los directorios deben coincidir. Para compilar, situados en el directorio de nuestro proyecto, debemos compilar así:

```
javac fuente1.java paquete\fuente2.java
paquete\subpaquete\fuente3.java
```

es decir, debemos compilar desde el directorio de nuestro proyecto y debemos poner, si hace falta, los path para llegar desde ahí a los fuentes. Esto generará los ficheros fuente1.class, paquete\fuente2.class y paquete\subpaquete\fuente3.class

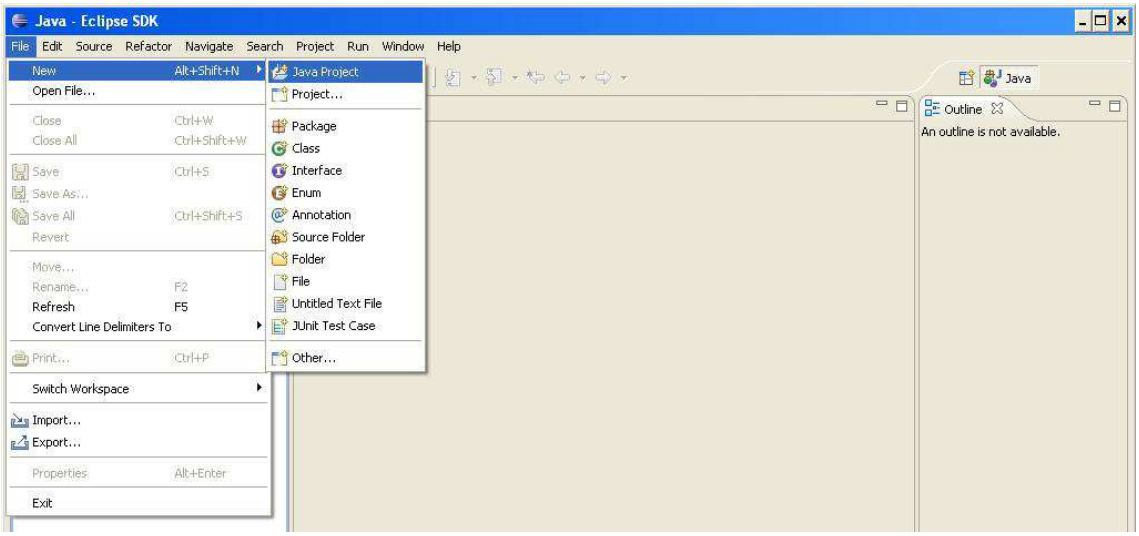
ECLIPSE

Eclipse es una entorno gráfico de desarrollo *open source* (código fuente disponible) basado en Java. El entorno Eclipse está estructurado en *perspectivas*. Cada *perspectiva* muestra las funcionalidades del entorno orientadas a una tarea concreta. Por ejemplo, la perspectiva *Java* (por defecto) resulta útil para escribir código fuente y ejecutar los programas, mientras que la perspectiva *Debug* resulta útil para la depuración del código. La perspectiva actual de Eclipse se puede cambiar seleccionando la opción del menú *Window >> Open perspective.*

Creación de un proyecto JAVA en ECLIPSE

Eclipse gestiona los desarrollos Java mediante proyectos. Para crear un proyecto nuevo son necesarios los siguientes pasos:

1. Seleccionar en el menú *File >> New >> Java Project*
2. Rellenar los datos del proyecto. Para un proyecto java sencillo como los que vamos a realizar en la asignatura con rellenar el nombre del proyecto es suficiente.
3. Una vez introducidos los datos del proyecto, pulsar sobre el botón *Finish*.

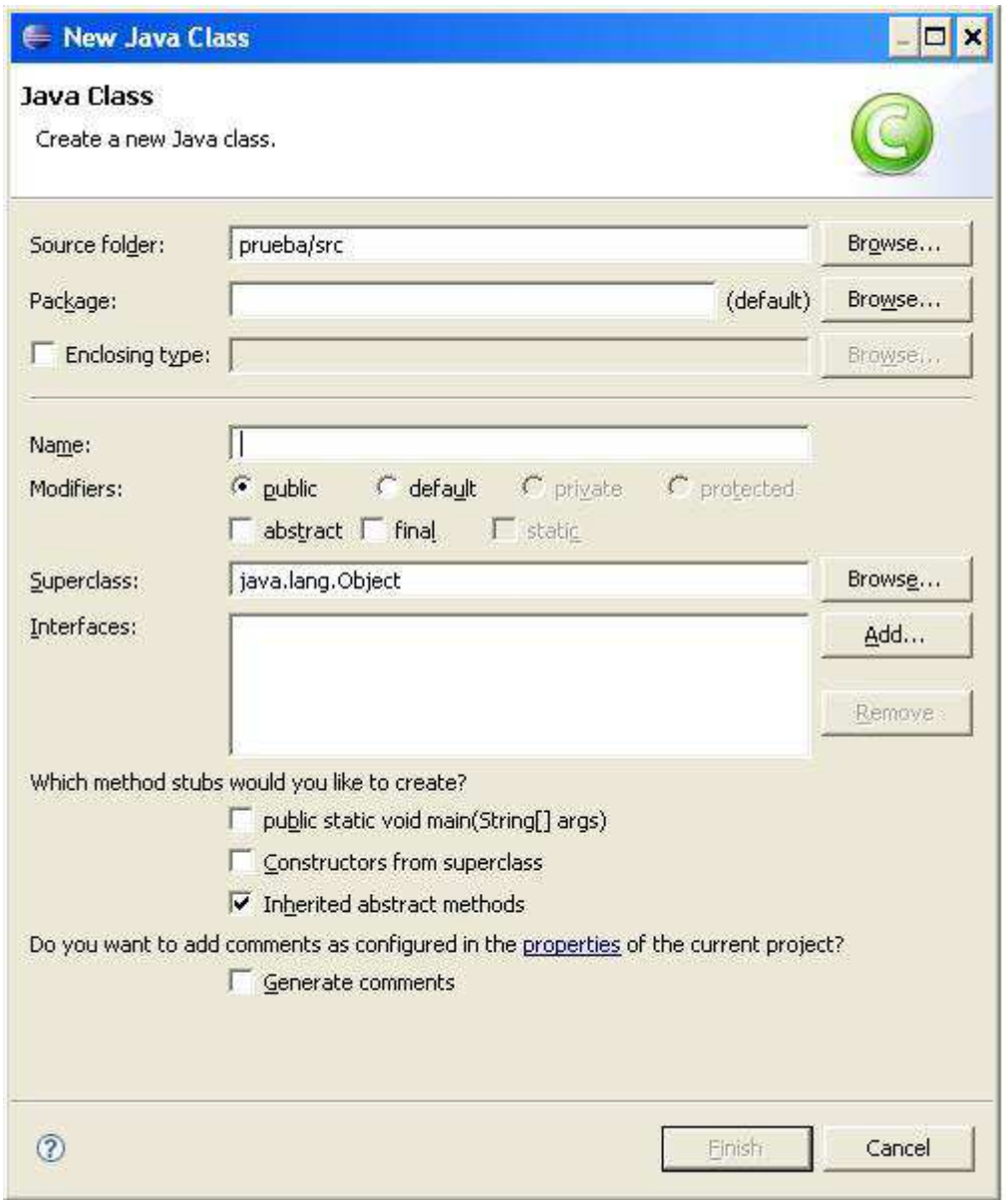


En este momento Eclipse crea una carpeta para el proyecto en el directorio de trabajo (*workspace)* con los ficheros *.project* y *.classpath*. Estos ficheros no contienen código fuente Java y únicamente son de utilidad para Eclipse. Es posible cambiar el directorio de trabajo seleccionando en el menú *File >> Switch WorkSpace*.

Creación de una CLASE en ECLIPSE

Para crear una clase Java que pertenezca al proyecto hay que seguir los pasos:

1. Seleccionar en el menú la opción *File>> New >> Class.*
2. Darle nombre a la clase en el diálogo de creación de clases que aparece a continuación.
3. Seleccionar el botón Finish.

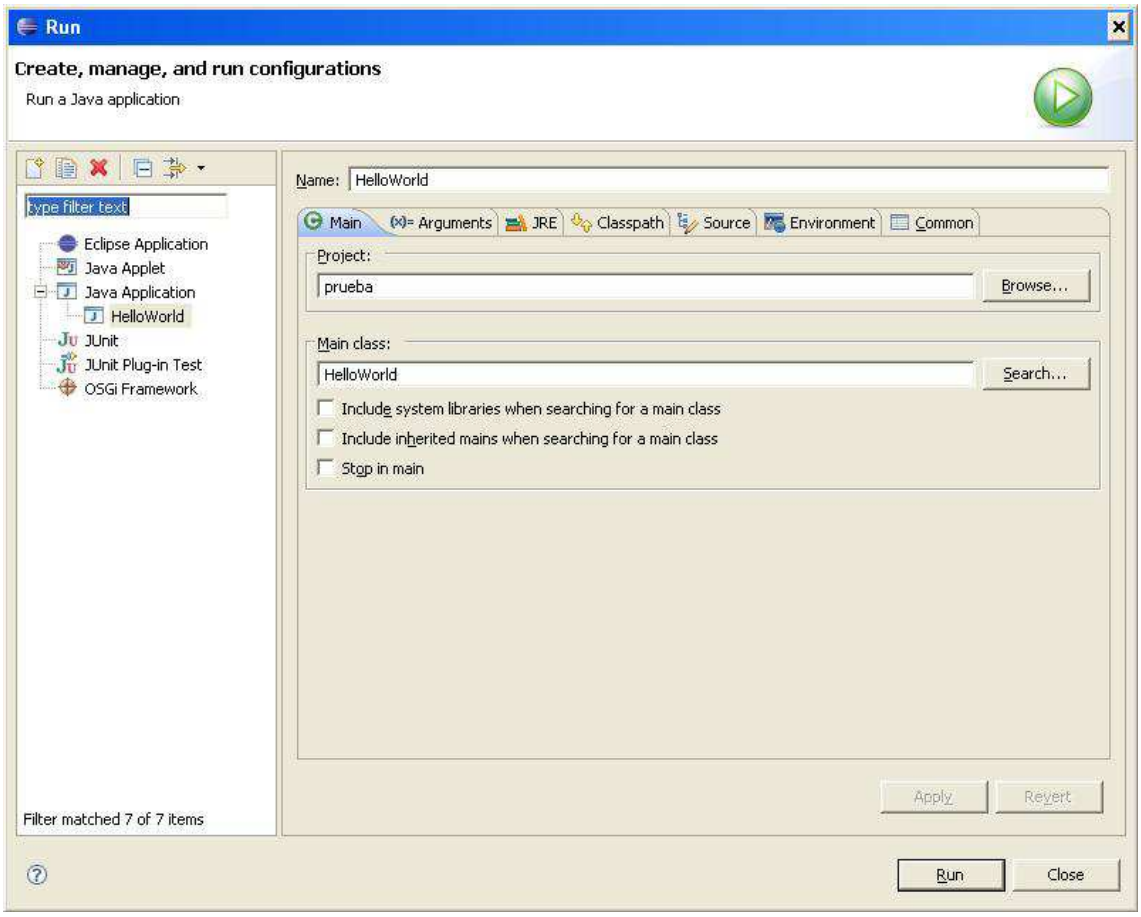


Inmediatamente Eclipse crea un fichero de nombre <nombre_de_la_clase>.java con el código correspondiente al esqueleto de la clase. A continuación se puede rellenar el resto del código de la clase.

Ejecución de una CLASE con ECLIPSE

Los programas se pueden ejecutar dentro de la perspectiva *Java* de Eclipse, seleccionando en el menú la opción*: Run >> Run*. En este momento Eclipse mostrará una la consola con el resultado de la ejecución.

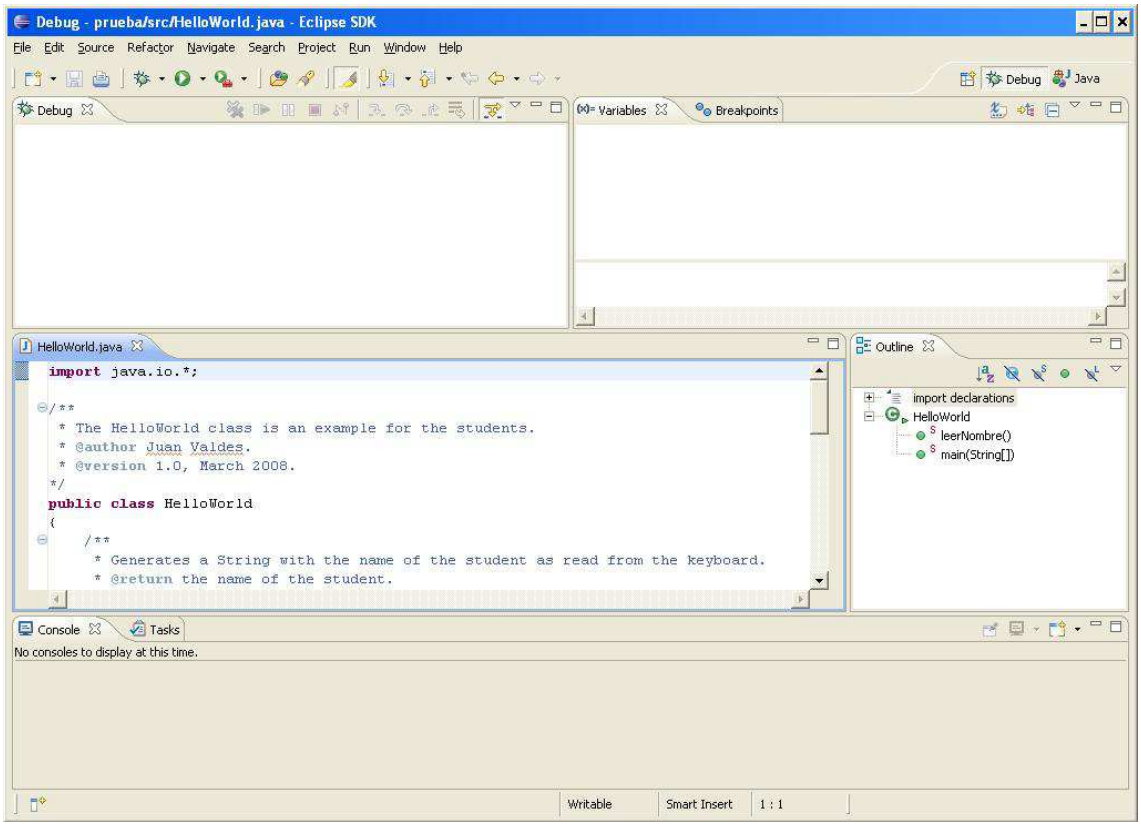
Además, Eclipse permite cambiar los argumentos con los que se llama al método *main* de la clase ejecutada seleccionando en el menú la opción *Run >> Open Run Dialog.* En la pestaña *Arguments* se puede introducir la lista de argumentos apropiada.



Depuración de programas con ECLIPSE

La perspectiva *debug* de Eclipse permite la depuración de programas. Esta perspectiva se lanza seleccionando en el Menú la opción *Window >> Open perspective >> Debug.* En la perspectiva *debug* la ejecución de un programa se puede inspeccionar de dos formas distintas:

1. Interrumpiendo la ejecución del programa en un punto de ruptura: La opción del menú *Run >> Toggle BreakPoint* permite especificar la línea de código en la que queremos que la ejecución se detenga. De forma que al lanzar posteriormente el depurador, *Run >> Debug,* la ejecución del programa se detendrá en el primer punto de ruptura.
2. Ejecutando el programa paso a paso mediante las opciones *Run >> Sep Into* o *Run >>StepOver*.



Ejercicios Java

Tabla de contenido

¿Cómo hacer los ejercicios?

Todos los ejercicios y prácticas aquí contenidas poseen un código único de fácil interpretación que los describe.
Por ejemplo:

EJ1.3-JAV-OP se refiere al ejercicio 3 del tema 1 (EJ1.3), perteneciente a la categoría Java (JAV) y es de carácter Optativo (OP).

En el caso de las prácticas, que son conjuntos de ejercicios que engloban varias categorías, son todas obligatorias.

Ejercicio

El primer número indica el tema, en el ejemplo anterior 1. Separado por un punto (.) se encuentra otro número perteneciente al ejercicio. Este código es único.

Categoría

La siguiente tabla ilustra las categorías recogidas y sus códigos:

Categoría	Código
Arrays	ARR
Bucles	BUC
Cadenas	CAD
Clases Anidadas	ANI
Clases Abstractas	ABS
Colecciones	COL
Conversiones	CON
Entrada y Salida	IO
Excepciones	EXC
Fechas	FEC
Herencia	HER
Instrucciones de control	INC
Interfaces	INT
Java	JAV
Nomenclatura	NOM
Operadores	OPE
Programación Orientada a Objetos	POO
Polimorfismo	POL
Sintaxis	SIN
Tipos Enumerados	ENU
Variables	VAR

Carácter del ejercicio

Indica la obligatoriedad del mismo, pudiendo ser:

- OB → Obligatorio
- OBE → Obligatorio Escrito
- OP → Optativo
- FH → Fuera de Horario

Tema 1: ¿Qué es Java?

EJ1.1-JAV-OP

¿Qué es Java?

a. Un Sistema Operativo
b. Un lenguaje de programación
c. Un compilador
d. Un programa

EJ1.2-JAV-OP

SDK es un acrónimo de:

a. Simple Development Kit
b. Software Developing Kat
c. Some Delivery Kit
d. Software Development Kit

EJ1.3-JAV-OP

¿Cuáles son las ediciones del JDK de Java?

a. J2EE, J2AA y J2II
b. J5EE, J5OO y J5UU
c. J2EE, J2SE y J2ME
d. J3XP, J3ME y J3VISTA

EJ1.4-JAV-OP

¿Cuál es un componente del SDK de Java?

a. Java
b. Javac
c. javadoc
d. Todos los anteriores

EJ1.5-JAV-OP

¿En qué sistema operativo se puede ejecutar una aplicación Java?

a. GNU/Linux
b. UNIX
c. Windows
d. En cualquiera que tenga instalado la JVM y/o JRE

EJ1.6-JAV-OP

El API (Interfaz de Programación de Aplicaciones) de Java

a. Se instala con la JVM
b. Se instala con el JDK

c. Se puede consultar online y descargase
d. Hay que comprarlo para acceder

EJ1.7-JAV-OP

¿Cuál es la extensión de un archivo de código fuente escrito en Java?

a. .class
b. .java
c. .jav
d. .j2se

EJ1.8-JAV-OP

¿Cuál es la extensión de un archivo compilado Java?

a. .jav
b. .byco
c. .class
d. .java

EJ1.9-JAV-OP

Las clases de Java se pueden empaquetar en archivos comprimidos, ¿Qué extensión tienen estos archivos?

a. .pack
b. .class
c. .jar
d. .tar

EJ1.10-JAV-OP

¿Qué popular algoritmo de compresión utilizan los paquetes de java?

a. El mismo que los archivos RAR (Roshal Archive)
b. El mismo que los archivos ZIP
c. El mismo que los archivos Ark
d. El mismo que los archivos 7z

Tema 2: POO

EJ2.1-POO-OP

De las siguientes, ¿Cuál es una ventaja de la Programación Orientada a Objetos?

a. Fomenta la reutilización y extensión del código.
b. Relaciona el sistema al mundo real.
c. Facilita el trabajo en equipo.
d. Todas las anteriores.

EJ2.2-POO-OP

¿Qué es una clase?

a. Una plantilla para la creación de objetos.
b. Un prototipo que define las propiedades y los métodos comunes a múltiples objetos de un mismo tipo.
c. a y b son correctas.
d. Ninguna de las anteriores.

EJ2.3-POO-OP

¿Qué es un objeto?

a. Una instancia de una clase.
b. Una función.
c. a y b son correctas.
d. Ninguna de las anteriores.

EJ2.4-POO-OP

Una clase está compuesta de:

a. Atributos.
b. Métodos.
c. a y b son correctas.
d. Ninguna de las anteriores.

EJ2.5-POO-OP

¿Para qué sirve el polimorfismo?

a. Permite tener métodos con el mismo nombre en distintas clases.
b. Permite enviar un mismo mensaje a objetos de clases diferentes.
c. a y b son correctas.
d. Ninguna de las anteriores.

EJ2.6-POO-OP

La herencia, en programación, es:

a. Una característica únicamente presente en Java.

b. Una característica de la Programación Orientada a Objetos.
c. Una característica de la Programación Estructural.
d. Una característica del lenguaje ensamblador.

EJ2.7-POO-OP

Mediante la herencia una clase hija adquiere de su clase padre:

a. Atributos, si son públicos o protegidos.
b. Métodos, si no son privados.
c. a y b son correctas.
d. Ninguna de las anteriores.

EJ2.8-POO-OP

Los objetos se comunican entre sí mediante:

a. Un archivo temporal.
b. Memoria principal.
c. Mensajes.
d. Correos electrónicos.

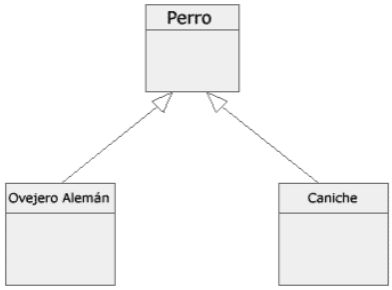
EJ2.9-POO-OP

Para enviar un mensaje se necesita:

a. El objeto al que se le va a enviar el mensaje.
b. El nombre del método que se debe ejecutar.
c. Los parámetros necesarios para el método en cuestión.
d. Todos los anteriores.

EJ2.10-POO-OP

¿Cuál sería la clase padre en el siguiente ejemplo?



a. *Perro*
b. Ovejero Alemán
c. *Caniche*
d. Ninguna de las anteriores

Tema 3: Preparando el entorno

EJ3.1-IO-OB

Implementar un programa que escriba en pantalla varias líneas de texto, combinando los métodos System.out.print y System.out.println. La salida podría ser la siguiente:

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

EJ3.2-SIN-OB

Escribe el siguiente programa, compílalo y corrige los errores que salgan.

```
1  public class JavaErrores;{
2          public static void main ( String [] args ){
3                  System.out.println ('He encontrado todos los errores.')
4          }
5  }
```

¿Cuántos errores has encontrado? Di número de línea y como lo has solucionado.

EJ3.3-JAV-OP

Si estamos trabajando con una versión de java igual o inferior a la 1.4 y nos encontramos con los siguientes programas, ¿Cuál sería el correcto si ejecutamos ambos?

HolaMundo1

```
1  public class HolaMundo1 {
2          public static void main (String... args)   {
3                  System.out.println("Hola mundo1!!");
4          }
5  }
```

HolaMundo2

```
1  public class HolaMundo2 {
2          public static void main (String[] args) {
3                  System.out.println("Hola mundo2!!");
4          }
5  }
```

a. *HolaMundo1*
b. *HolaMundo2*
c. a y b son correctas.
d. Ninguna de las anteriores.

EJ3.4-NOM-OP

De acuerdo con las convenciones de nomenclatura en java, ¿Cuáles de las siguientes palabras se consideran correctas como nombres de clases o interfaces?

a. Unaclase
b. 1Clase
c. UnaClase
d. Animales
e. aniMales
f. perro
g. Gato
h. CienPies
i. LaClaseQueEstoyProbandoAhoraMismo

EJ3.5-NOM-OP

De acuerdo con las convenciones de nomenclatura en java, ¿Cuáles de las siguientes palabras se consideran correctas como nombres de objetos, métodos o variables?

a. unavariable
b. 1metodo
c. Unobjeto
d. unObjeto
e. mi-variable
f. esa_Variable
g. elObjetoQueEstoyInstanciando
h. ElmetodoMasLargoDelMundo
i. unaVariableMuyMuyLarga

Tema 4: Sintaxis del lenguaje

EJ4.1-NOM-OP

De los siguientes identificadores cuales no son válidos (no compilaría la aplicación) y porque.

Nota: no tienen porque seguir las convenciones de nomenclatura, solo los que no compilarían.

a. $identificador
b. Un Identificador
c. 1identificador
d. _elDia11
e. Año
f. Unaclase
g. El%Descontado
h. dollar$
i. cuanto?

EJ4.2-COM-OP

De los siguientes comentarios cuales son válidos:

a. /esto es un comentario de una línea
b. //esto es un comentario
c. De varias líneas
d. /*Comentario de
e. Varias líneas*/
f. //Comentario de
g. //Varias líneas*/
h. //Comentario de 1 línea

EJ4.3-VAR-OB

Implementar un programa en el que se declare e inicialice una variable de cada uno de los tipos primitivos y escriba en pantalla frases del tipo:

El valor de la variable “<tipo>” es: <???>

EJ4.4-VAR-OP

Rangos de los tipos básicos sobre el ejercicio anterior (no cambie nada, ponga las nuevas instrucciones a continuación del último System.out.println), para las variables que contienen datos de tipo numérico, pruebe a dar nuevos valores que sobrepasen los rangos de cada uno de los tipos. Imprima los resultados por pantalla, ¿qué ocurre en cada caso?

EJ4.5-VAR-OP

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él declare de una sola vez 3 variables del mismo tipo básico (cualquiera de ellos). Declare también de una sola vez cuatro variables de otro tipo, dando valor a la primera y a la última en la misma declaración. A continuación en otra línea dé valor a la segunda variable, y seguidamente para dar valor a la tercera combine las variables anteriores. Imprima las variables por pantalla. Cambie el valor de la primera variable, imprima por pantalla el valor de la tercera, ¿Ha cambiado? ¿Por qué?

EJ4.6-VAR-OB

Cree una clase y dentro de ella un método *main*. Dentro de él declare una variable de cualquier tipo, a continuación vuelva a declarar otra vez la misma variable con el mismo tipo. ¿Es posible? ¿Por qué? ¿Y si declaramos la variable con un tipo distinto?
Introduzca la declaración de variable anterior entre dos llaves { }. ¿Es posible ahora volver a declararla? ¿Por qué?
Dentro de las llaves dé valor a la variable. Imprímala fuera de las llaves. ¿Qué ocurre? ¿Por qué?

EJ4.7-VAR-OB

En el ejercicio anterior, ponga final delante de la declaración de la primera variable de la segunda tanda, ¿ocurre algo? ¿Por qué?
Quite lo anterior y ponga final delante de la tercera variable, ¿Ocurre algo? ¿Por qué?
Cree una constante de tipo String y asígnele valor. En la línea siguiente asígnele un nuevo valor, ¿es posible hacerlo?

EJ4.8-VAR-OP

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él teclee lo siguiente:

```
char a;
a = '\\\';
System.out.println(a);
```

¿Cuál es el resultado del código anterior? ¿Por qué?
¿Es posible crear un char con más de un carácter (ejemplo char b = 'hola')?

EJ4.9-VAR-OP

Cree dos variables de cualquiera de los tipos básicos, dé valor a la primera y a continuación iguale la segunda a la primera. Imprima por pantalla la segunda. Cambie el valor de la primera y vuelva a imprimir la segunda, ¿Cambia la segunda de valor? ¿Por qué? Haga lo mismo para dos variables de la clase String.

EJ4.10-OPE-OB

A partir de una variable *num1* con el valor 12 y una variable *num2* con el valor 4, utiliza nuevas variables en las cuales se almacene la suma, resta, división y multiplicación de *num1* y *num2* y muéstrelas por pantalla.
A continuación, realice la operación: (7+14) / 7. Escríbela con y sin paréntesis y observa el resultado. ¿Qué conclusiones sacas?

EJ4.11-OPE-OP

Implementar un programa que asigne 10 a una variable llamada *num1*, la muestre por pantalla, asigne 7 a un variable llamada *num2*, la muestre por pantalla y luego muestre la suma de ambas variables sin utilizar ninguna nueva variable y sin modificar *num1* y *num2*.

EJ4.12-OPE-OP

Implementar un programa que asigne 7 a una variable llamada *numero*, muéstrala por pantalla luego suma 1 a dicha variable, vuelve a mostrarla por pantalla, vuelve a restarle uno y vuelve a mostrar su valor por pantalla.

EJ4.13-OPE-OP

Realizar el ejercicio anterior pero con un sistema distinto de sumar y restar uno a la variable.

EJ4.14-OPE-OB

Intercambia el valor de dos variables llamadas *var1* y *var2* que tenga inicialmente (antes del intercambio) los valores 2 y 5 respectivamente. Muestra mensajes en pantalla con el valor que tiene cada variable antes y después del intercambio.
Nota: Para Realizar este intercambio de variables no se pueden hacer asignaciones directas de valores, si hace falta utiliza una nueva variable.

EJ4.15-OPE-OBE

¿Cuál es el resultado del siguiente programa?

```
1  class Ejemplo {
2          public static void main(String [] args) {
3                  int a=1, b=4, c=2, d=1;
4                  int x=a+b/c+d;
5                  System.out.print("x ="+ x);
6          }
7  }
```

EJ4.16-OPE-OBE

Suponga que *b* es una variable lógica (boolean). ¿Cuál es el resultado de las siguientes expresiones?

a) b==true

b) b=true

EJ4.17-VAR- OBE

¿Cuál es el resultado del siguiente programa?

```
1  class Alcance {
2          public static void main(String [] args) {
3                  int i=3;
4                  {
5                  int j=4;
6                  }
7                  System.out.println("j: "+j);
8                  System.out.println("i: "+i);
9          }
10 }
```

EJ4.18-OPE-OBE

Indique cuál es la salida del siguiente programa:

```
1  class Ejercicio {
2          public static void main (String[] args) {
3                  char probador;
4                  probador = 'c';
5                  System.out.println("probador:" + probador);
6                  ++probador;
7                  System.out.println("probador:"+probador);
8                  System.out.println("probador:"+ probador++ + probador+probador-- + probador);
9          }
10 }
```

EJ4.19-OPE-OBE

Implementar un programa que sirva para comprobar que la tabla de los operadores lógicos vista en clase es correcta. Para ello se han de evaluar las operaciones lógicas sobre dos variables de tipo booleano x e y, a las que se les va asignando cada vez un valor lógico distinto.

EJ4.20-OPE-OBE

Indique cuál es la salida del siguiente programa:

```
1  class Ejercicio {
2          public static void main(String[] args) {
```

```
3                   int suma=30;
4                   System.out.println (suma++ + " " + ++suma +" " + suma + " " + suma--);
5                   System.out.println(suma);
6           }
7   }
```

EJ4.21-OPE-OBE

¿Cuál es el resultado del siguiente programa?

```
1   class Ejercicio{
2           public static void main(String [ ] args) {
3                   int var=1;
4                   boolean r,s,t,v;
5                   r=(var>1) && (var++ <100);
6                   s=(100 < var) && ( 150 > var++);
7                   t=(100 == var) || (200 > var++);
8                   v=(100 == var) || (200 > var++);
9                   System.out.println(r +" " + s +" " +t + " " + v);
10          }
11  }
```

EJ4.22-OPE-OBE

¿Cuál es el resultado de evaluar las siguientes expresiones si suponemos que, inicialmente, x vale 1?

a. (x > 1) & (x++ < 10)
b. (1 > x) && (1 > x++)
c. (1 == x) | (10 > x++)
d. (1 == x) || (10 > x++)
e. (++x) + x;
f. x + (++x)

EJ4.23-OPE-OBE

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él declare dos variables de tipo *long*, dé valor a la primera y haga que la segunda sea igual a la primera autoincrementada (autoincremento delante). Imprima la segunda.
A continuación haga lo mismo pero poniendo el autoincremento detrás, ¿Se produce algún cambio? ¿Por qué?
Haga lo mismo para el operador autodecremento.

EJ4.24-OPE-FH

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él declare tres variables para cada uno de los cuatro tipos enteros y otras tres de *char*. Para cada uno de los tipos, dé valores adecuados a las dos primeras y a la tercera

asígnele el valor de operar las dos anteriores con todos los operadores aritméticos de Java (suma, resta, etc.). Imprima los resultados por pantalla.
¿Se produce algún fallo? ¿Por qué? ¿Cómo se podrían resolver?
Nota: una vez descubiertos los fallos, para que el programa pueda compilar, en lugar de borrar las líneas en las que hay fallos, conviértalas en comentarios usando //.

EJ4.25-OPE-OBE

Si declaramos tres variables de tipo *int*. Se le asigna valor 5 a la primera y valor 0 a la segunda. Si se igual la tercera sea a la segunda dividida por la primera. ¿Se produce algún fallo? ¿Por qué? ¿Cambia el resultado si las variables en lugar de ser de tipo *int* son de cualquier otro tipo entero?

EJ4.26-OPE-OBE

Si declaramos una variable de tipo *char* con un valor adecuado. Al ejecutar, *System.out.println(variable+1)*. ¿Qué ocurre?
¿Y si ponemos *System.out.println(variable++)*?

EJ4.27-OPE-OP

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él declare tres variables de tipo String. Dé valor a las dos primeras y haga que la tercera sea igual a primera+segunda. Imprima la tercera. ¿Qué ocurre? ¿Y si hacemos tercera= primerasegunda?

EJ4.28-OPE-FH

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él teclee lo siguiente:

```
1  int a,b;
2  float c=3;
3  boolean r,s,t,u,v,w,x;
4  a = 3;
5  b = 8;
6  r = a == 0;
7  s = a != 0;
8  t = a <= b;
9  u = b >= a;
10 v = b > a;
11 w = b < a;
12 x = c == 3.0;
13 System.out.println(“r:” + r);
14 System.out.println(“s:” + s);
15 System.out.println(“t:” + t);
16 System.out.println(“u:” + u);
17 System.out.println(“v:” + v);
18 System.out.println(“w:” + w);
19 System.out.println(“x:” + x);
```

¿Cuál es el resultado del código anterior? ¿Por qué?

EJ4.29-OPE-FH

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él teclee lo siguiente:

```
1  int a,b;
2  boolean r,s,t;
3  a = 3;
4  b = 8;
5  r = a == 0 | b >= a;
6  s = a != 0 & b < a;
7  t = a <= b ^ b > a;
8  System.out.println("r:" + r);
9  System.out.println("s:" + s);
10 System.out.println("t:" + t);
```

¿Cuál es el resultado del código anterior? ¿Por qué?

EJ4.30-OPE-FH

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él teclee lo siguiente:

```
1  int a=5, b=3;
2  boolean r=true, s=false;
3  a+=b+8*b;
4  r&=s;
5  System.out.println("a:" + a);
6  System.out.println("b:" + b);
7  System.out.println("r:" + r);
8  System.out.println("s:" + s);
```

¿Cuál es el resultado del código anterior? ¿Por qué?

EJ4.31-OPE-FH

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él teclee lo siguiente:

```
1  int a=5,b=3,c=20,d=20;
2  c-=++a/b-3+a%b
3  d-=++a/(b+3-4*a)%b
4  System.out.println("c:" + c);
5  System.out.println("d:" + d);
```

¿Cuál es el resultado del código anterior? ¿Por qué?

EJ4.32-CON-OBE

¿Cuál es el resultado del siguiente programa?

```
1   class Ejercicio {
2           public static void main(String [] args) {
3                   int a=1, b=2, c=3, d=1;
4                   float r, s=(float)3.0;
5                   r=a+b/c+d/a;
6                   s=r-s;
7                   r=(long) s;
8                   r=++r;
9                   System.out.println(r);
10          }
11  }
```

EJ4.33-CON-OB

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él declare y dé valor a una variable para cada tipo básico, incluyendo String. A continuación, asigne sucesivamente una variable a todas las demás. Complete la siguiente tabla con las asignaciones válidas (ponga "SI" si la una variable de la fila se puede asignar a una de la columna y "NO" en caso contrario)

Tipo	byte	Short	int	long	float	double	char	boolean	String
Byte	SI								
Short		SI							
int			SI						
long				SI					
float					SI				
double						SI			
char							SI		
boolean								SI	
String									SI

EJ4.34-CON-OB

Cree una clase denominada EjXX copiando la anterior, fuerce ahora las conversiones entre tipos usando el casting adecuado y vuelva a rellenar la tabla anterior.
¿Siguen quedando casillas con "NO"?, en caso afirmativo, ¿por qué?

EJ4.35-CON-OP

Cree una clase denominada EjXX. Dentro de ella cree un método *main* y dentro de él teclee lo siguiente:

```
1  char a;
2  int b;
3  short c;
4  long d;
5  float e;
6  double f;
7  System.out.println("asignamos variable char a enteros");
8  a = '4';
9  b = a;
10 c = a;
11 d = a;
12 System.out.println("a:" + a);
13 System.out.println("b:" + b);
14 System.out.println("c:" + c);
15 System.out.println("d:" + d);
16 System.out.println("asignamos variable double a float");
17 f = 1e200;
18 e = f;
19 System.out.println("f:" + f);
20 System.out.println("e:" + e);
21 System.out.println("asignamos variable float a int");
22 e = 1234.5678;
23 b = e;
24 System.out.println("e:" + e);
25 System.out.println("b:" + b);
```

¿Cuál es el resultado del código anterior? ¿Por qué?

EJ4.36-INC-OB

Definir dos variables *primerNumero* y *segundoNumero* e implementar un programa que asigne un valor a cada una y obtenga el mayor de los dos, mostrándolo en un mensaje en pantalla. Para ello utiliza la construcción *if* preguntando por el valor de dichas variables

EJ4.37-INC-OBE

¿Cuál es el resultado del siguiente programa?

```
1  class Ejercicio {
2          public static void main(String [] args) {
3                  int s,x=0;
4                  switch (x) {
5                          case 0:
6                                  s=0;
7                          default:
8                                  if (x<0)
9                                          s=-1;
10                                 else
11                                         s=1;
```

```
12                      }
13                      System.out.println(s);
14              }
15      }
```

EJ4.38-INC-OB

Implementar el código necesario para conocer la estación del año según el mes. Para ello, defina una variable entera que haga referencia al número del mes. Utilizar una instrucción de control *switch* para que en caso de estar en Enero, Febrero o Marzo, imprima por pantalla "invierno", Abril, Mayo o Junio, imprima "primavera", Julio, Agosto, Septiembre imprima "verano", Octubre, Noviembre, Diciembre imprima "otoño".

EJ4.39-INC-OP

Crear una clase para implementar el funcionamiento básico de un cajero automático. Para ello, defina 2 variables de tipo entero, una llamada *saldoEnCuenta*, otra llamada *cantidadASacar*. Dé valores aleatorios a ambas variables. Si el saldo en cuenta es mayor o igual que la cantidad a sacar, imprima por pantalla: "Petición aceptada" y el saldo resultante en cuenta. En cambio, si el saldo en cuenta es menor que la cantidad demandada imprima por pantalla: "Saldo insuficiente".

EJ4.40-INC-OP

Implementar un programa que dado un número del 1 al 7, y que diga a que día de la semana corresponde (1- lunes, 2- martes,…).

EJ4.41-INC-OB

Implementar un programa que inicialice dos números enteros y diga si son iguales o no lo son. A continuación, el programa ha de mirar cual de los dos es el mayor, y después mirar si este número mayor es divisible por el menor; en caso que lo sea ha de restar el mayor del menor, en caso contrario los ha de sumar.

EJ4.42-INC-OP

Implementar un programa que dados cuatro números A, B, C y D; el programa ha de sumarlos si A/B > C/D sino, ha de sumar sólo B y D. En caso que B=0, o D=0 el programa ha de mostrar un mensaje avisando que no se puede dividir por 0; en tal caso no ha de Implementarse ninguna operación.

EJ4.43-BUC-OBE

¿Cuántas veces se ejecutaría el cuerpo de los siguientes bucles *for*?

a. for (i=1; i<10; i++)
b. for (i=30; i>1; i-=2)
c. for (i=30; i<1; i+=2)

d. for (i=0; i<30; i+=4)

EJ4.44-BUC-OBE

Ejecute paso a paso el siguiente bucle (no es necesario compilarlo):

```
1  c = 5;
2  for (a=1; a<5; a++)
3          c = c – 1;
```

EJ4.45-BUC-OP

Muestra los 10 primeros números naturales por pantalla mediante una estructura *for*.

EJ4.46-BUC-OB

Muestra los números impares que hay entre 1 y 10 por pantalla mediante una estructura *for*.

EJ4.47-BUC-OP

Muestra los 10 primeros números naturales impares por pantalla mediante una estructura *for*.

EJ4.48-BUC-OP

Desarrollar un programa en el que, usando un bucle *for*, se escriba en pantalla una tabla de conversión de grados Fahrenheit a Celsius para valores desde 0 hasta 300 a intervalos de 20. (0, 20, 40, etc.). La regla de conversión es la siguiente: °C = (5/9)(°F - 32).

EJ4.49-BUC-FH

Desarrollar un programa en el que, usando bucles *for*, se escriban en pantalla todos los pares [i, j] que representan una ficha de dominó en la que 0 <=i <=j<= 6.

EJ4.50-BUC-OB

Escribir un programa utilizando bucles *for* que imprima por pantalla lo siguiente:

```
1
12
123
1234
1234
123
12
```

1

EJ4.51-BUC-FH

Definir un array bi-dimensional para llevar una agenda semanal, que represente los días de la semana y las 24 horas de cada día. Utilice bucles *for* anidados para inicializar la agenda a: “No tengo planes” y muestre el resultado por pantalla.

EJ4.52-BUC-OP

Utilizar la sentencia de control adecuada para imprimir la tabla de multiplicar del número cinco. La salida debe tener el formato:

5x1 = 5
5x2 = 10
5x3 = 15
Etc

EJ4.53-BUC-FH

Imprima por pantalla las horas en formato hh:mm que hay entre las 17:15 y las 18:34 (incluida) utilizando una sentencia repetitiva.

EJ4.54-BUC-OBE

El siguiente fragmento de programa pretende sumar los enteros de 1 a n (ambos inclusive) almacenando el resultado en la variable *sum*. ¿Es correcto el programa? Si no lo es, indique por qué y qué habría que hacer para solucionarlo.

```
1  i=0;
2  sum=0;
3  while (i<=n) {
4          i=i+1;
5          sum=sum+i;
6  }
```

EJ4.55-BUC-OB

Implementar un programa en el que, usando un bucle *while*, se escriba en pantalla una tabla de conversión de euros a pesetas para valores desde 1 hasta 10 €. (1 € = 166.386 pts).

EJ4.56-BUC-OP

Definir un array de enteros de 100 elementos. Utilice un bucle de tipo *while* para inicializar el array con los números del 1 a 100. A continuación, utilice un bucle de tipo for para mostrar por pantalla el contenido del array.

EJ4.57-BUC-FH

Implementar un programa en el que, usando un bucle *while*, se escriban en pantalla los 51 primeros valores de la sucesión de Fibonacci, definida por recurrencia como sigue:

f0 = 0
f1 = 1
fn+2 = fn+1 + fn

EJ4.58-BUC-FH

Escriba una función que, a partir de los dígitos de un ISBN, calcule el carácter de control con el que termina todo ISBN. Para calcular el carácter de control, debe multiplicar cada dígito por su posición (siendo el dígito de la izquierda el que ocupa la posición 1), sumar los resultados obtenidos y hallar el resto de dividir por 11. El resultado será el carácter de control, teniendo en cuenta que el carácter de control es 'X' cuando el resto vale 10.

EJ4.59-ARR-OBE

Si declaramos un array de alguno de los tipos enteros, otro de los reales, otro de *char*, otro de *boolean* y otro de *String*. ¿Qué ocurriría si intentamos utilizar alguno de los elementos sin haberle dado valor antes?

EJ4.60-ARR-OBE

Si igualamos dos elementos (ejemplo a[5]=a[3]) de un array. Cambiamos el segundo elemento de valor, al imprimir los dos otra vez ¿cambia también el primero? ¿Por qué?

EJ4.61-ARR-OBE

Si intentamos cambiar la longitud de un array poniendo por ejemplo *array.length* = 5, ¿Qué ocurre? ¿Por qué?

EJ4.62-ARR-OBE

Si creamos dos arrays, rellenamos el primero e igualamos el segundo al primero. Al cambiar el valor de un elemento del primer array. ¿Cambia el elemento correspondiente del segundo? ¿Por qué?

Y si hacemos lo mismo pero en lugar de igualarlos usamos S*ystem.arraycopy(),* ¿hay alguna diferencia?

EJ4.63-ARR-OP

Implementar el código necesario para representar un tablero de ajedrez con la posición inicial de todas las piezas. Imprimir por pantalla el tablero, implementar el movimiento de una de las piezas e imprimir por pantalla la situación final de tablero. Representar:

• Casillas vacías: espacios en blanco y símbolos #, para casillas blancas y negras respectivamente.

• Casillas ocupadas: inicial del nombre de la pieza que la ocupa (ej. T para la torre).

EJ4.64-ARR-OB

Definir un array bidimensional de 3 x 4 de tipo String e inicializarlo. Imprimir por pantalla sus dimensiones utilizando *length*.

EJ4.65-ARR-OP

Implementar el funcionamiento básico de una agenda. Para ello, definir un array bidimensional llamado año de longitud 12 en el que cada uno de sus elementos represente un mes (array de Strings). La longitud de cada uno de estos arrays unidimensionales dependerá del número de días que tenga el mes correspondiente en un año no bisiesto, es decir, el primer elemento tendrá longitud 31, el segundo 28, etcétera.

EJ4.66-ARR-OBE

¿Qué imprime el siguiente programa?

```
1   class Ejercicio {
2           public static void metodo1(int [] m1, int [] m2) {
3                   m1[1]=4;
4                   m2[1]=m1[3]+m2[2];
5           }
6           public static void main(String [] args) {
7                   int[] matriz1={9,1,3,5};
8                   int [] matriz2=matriz1;
9                   matriz2[3]=4;
10                  matriz1[1]=3;
11                  metodo1(matriz1, matriz2);
12                  System.out.print(matriz1[1]+" "+matriz2[2]+" ");
13          }
14  }
```

Tema 5: Clases de uso general

EJ5.1-IO-OB

Implementar un programa que escriba en pantalla la frase “Mi nombre es xxx”, siendo xxx un parámetro que llega mediante los argumentos del *main()*.

EJ5.2-IO-OP

Implementar un programa que reciba el año de nacimiento por parámetros e imprima por pantalla los años que tengas.

Por ejemplo, si estamos en 2010, al ejecutar:
“*java CalcularEdad 1985*” tiene que devolver “*Tienes 25*”

EJ5.3-IO-OBE

Ante el siguiente programa,

```
1  public class PasoParametros {
2          public static void main (String[] args) {
3                  System.out.println("Hola:  " +args[1]+".");
4          }
5  }
```

Si lo ejecutamos de la siguiente manera:

```
>java PasoParametros
```

¿Qué salida obtendremos al ejecutarlo?

a. Hola “args[1]”.
b. Hola .
c. Nada
d. Error

EJ5.4-IO-OBE

Si el ejercicio anterior lo ejecutamos de la siguiente manera:

```
>java PasoParametros jose pepe
```

¿Qué salida obtendremos?

a. Hola jose pepe.
b. Hola jose.
c. Hola pepe.
d. Error

EJ5.5-IO-OBE

Si el ejercicio anterior lo ejecutamos de la siguiente manera:

```
>java PasoParametros jose pepe
```

¿Cómo habría que reescribir la línea 3 del código del ejercicio 3.6 para que se imprima “Hola: jose pepe.” por pantalla?

a. System.out.println("Hola: " +args[0]+” ” +args[1]+".");

b. System.out.printf("Hola: %s %s. ", args[0],args[1]);
c. System.out.print("Hola: "+args[0]+" "+args[1]+".");
d. Todas son correctas

EJ5.6-OPE-OB

Implementar un programa en el que se pidan al usuario dos datos de tipo entero y escriba en pantalla el cociente y el resto de la división entera entre ambos.

EJ5.7-OPE-OP

Realizar un programa que pida dos números enteros por pantalla y diga si el primero es divisible por el segundo. Para ello utilizar el operador % que devuelve el resto de la división.

EJ5.8-BUC-OB

Desarrollar un programa en el que se pida al usuario dos números enteros positivos, n y m, y, usando un bucle *for*, se escriba en pantalla el valor de n elevado a m.

EJ5.9-BUC-OP

Crear una clase que reciba como parámetro un número entero positivo y calcule su factorial.

EJ5.10-BUC-OP

Crear una clase que muestre por pantalla los divisores de un número recibido como parámetro.

EJ5.11-BUC-FH

Crear una clase que dada una hora calcule y muestre por pantalla la hora correspondiente al segundo siguiente. La hora se recibirá como parámetro en forma de array de tres posiciones, la primera para horas, la segunda para minutos y la tercera para segundos.

EJ5.12-BUC-FH

Crear un programa que dadas dos fechas cualesquiera que recibirá como parámetros (*día, mes, año*) de forma similar al problema anterior, calcule el número de días que hay entre ellas. Se deberán comprobar los años bisiestos.

EJ5.13-BUC-OP

Desarrollar un programa en el que se pida al usuario un valor entero positivo, *n* y, usando un bucle *while*, se escriba en pantalla el valor del factorial de *n*.

$n! = n \times (n - 1) \times \ldots \times 1$.

EJ5.14-BUC-OP

Desarrollar un programa en el que se pida al usuario un valor entero positivo, n y, usando un bucle *while*, se escriba en pantalla el valor de la raíz cuadrada entera de n.

EJ5.15-BUC-OB

Desarrollar un programa en el que se pida al usuario un valor entero positivo, n y, usando un bucle *while*, se escriban en pantalla todos los múltiplos de 7 menores o iguales que el número n.

EJ5.16-BUC-OP

Un hotel tiene 6 plantas, las 5 primeras tienen 100 habitaciones mientras la última, donde están las suites, sólo tiene 40. Cada una de las habitaciones puede estar ocupada o no por una persona. Crear un array que represente el conjunto de habitaciones del hotel y permita guardar información acerca de si una habitación está ocupada o no y quién (nombre y apellidos) la ocupa.
Implementar el código necesario para que al ejecutar la clase se reciba mediante parámetros el número de planta y de habitación y el programa devuelva si está ocupada o no, y en caso afirmativo quién la ocupa. Se deberá controlar que tanto la planta como la habitación estén en el rango permitido, imprimiendo "*número de habitación incorrecto*" en caso contrario.

EJ5.17-ARR-OB

Implementar un programa que escriba en pantalla línea a línea todos los datos proporcionados en la entrada, si no se proporciona ninguno deberá avisarse con un mensaje.

EJ5.18-ARR-OP

Implementar un programa que escriba en pantalla todos los datos proporcionados en la entrada en forma de vector, utilizando '[' y ']' como delimitadores y ',' como separador.

Ejemplo:

```
> java EjercicioXXX 1 2 3 4 5
[1,2,3,4,5]
```

EJ5.19-IO-OB

Realizar un programa que lea una línea por teclado y la vuelva a escribir en la pantalla.

EJ5.20-IO-OB

Realizar un programa que lleva a cabo las siguientes operaciones a partir de 3 variables enteras que lea por teclado, mostrando al final, por pantalla los resultados de las operaciones (a + b) * c

(a * b) + c
(a / b) + c
(a + b) / c
(a * b) / c
(a / b) * c

EJ5.21-IO-OB

Implementar un programa en el que se pida al usuario que introduzca un dato de tipo entero y otro de tipo real y, una vez hecho esto, se escriba en pantalla los valores introducidos, tal y como se indica en el ejemplo:

Introduzca un número entero: 7
Introduzca un número real: 3.45
El número entero proporcionado es 7
El número real proporcionado es 3.45

EJ5.22-IO-OP

Implementar un programa en el que se pidan al usuario tres datos de tipo entero, *a*, *b* y *c*, y escriba en pantalla la ecuación de segundo grado.

EJ5.23-IO-OP

Implementar un programa en el que se pida al usuario un valor real, *x*, y se escriba en pantalla el valor de la función f(x) = 1/(x2 - 1). Para los valores de *x* en los que la función f(x) no esté correctamente definida, se debe escribir un aviso en pantalla.

EJ5.24-IO-FH

Implementar un programa en el que se pidan al usuario las coordenadas de un punto del plano (x,y), e indique en pantalla si el punto se encuentra en el interior del círculo unidad (centro (0, 0) y radio 1), en el borde de dicho círculo o en el exterior del mismo.

EJ5.25-IO-OB

Implementar un programa en el que se pidan al usuario tres números reales, y se escriban en pantalla ordenados de menor a mayor.

EJ5.26-IO-FH

Una línea de autobuses cobra un mínimo de 20 € por persona y trayecto. Si el trayecto es mayor de 200 km el billete tiene un recargo de 3 céntimos por km adicional. Sin embargo, para trayectos de más de 400 km el billete tiene un descuento del 15 %. Por

otro lado, para grupos de 3 o más personas el billete tiene un descuento del 10 %. Con las consideraciones anteriores, escriba en Java un programa estructurado que lea por teclado la distancia del viaje a realizar, así como el número de personas que viajan juntas y que con ello calcule el precio del billete individual.

EJ5.27-ARR-OB

Desarrollar un programa en el que se pida al usuario un vector de números enteros y obtenga el máximo y mínimo de los valores incluidos en el vector.

EJ5.28-ARR-OP

Desarrollar un programa en el que se pida al usuario un vector de números enteros e indique en pantalla si los elementos de dicho vector están ordenados de menor a mayor o no.

EJ5.29-ARR-OP

Desarrollar un programa en el que se pida al usuario un vector de números enteros e indique en pantalla la media de todos sus elementos.

EJ5.30-ARR-OB

Desarrollar un programa en el que se pidan al usuario un vector de números enteros e indique en pantalla si dicho vector es capicúa, es decir, la secuencia de sus elementos es igual vista de delante hacia atrás y de detrás hacia delante.

EJ5.31-ARR-OP

Desarrollar un programa en el que se pida al usuario un vector de números enteros e indique en pantalla cuantos de dichos elementos son números impares.

EJ5.32-ARR-OP

Desarrollar un programa en el que se pida al usuario un vector de números enteros e indique en pantalla el número de ocurrencias de elementos repetidos.
Por ejemplo el vector [1, 2, 3, 1, 2, 1] tiene tres ocurrencias de elementos repetidos (dos 1 y un 2).

EJ5.33-ARR-OB

Desarrollar un programa en el que se pidan al usuario dos vectores de números enteros *v1* y *v2*, se construya el vector resultado de “concatenar” los vectores *v1* y *v2*, es decir, poner los elementos de *v2* a continuación de los de *v1* y, finalmente, se escriban en pantalla todos los elementos de la concatenación.

EJ5.34-ARR-OP

Desarrollar un programa en el que se pida al usuario dos vectores de números enteros, se construya un nuevo vector *v* que almacene la suma de ambos vectores y, finalmente, se escriban en pantalla todos los elementos de *v*. El vector suma se ha de ajustar al vector más largo proporcionado por el usuario, completando el más corto con ceros.
Por ejemplo, la suma de los vectores [1, 2, 3] y [1, 2, 3, 4, 5] es [2, 4, 6, 4, 5].

EJ5.35-ARR-FH

Desarrollar un programa en el que se construya una matriz de tamaño 3×3 de números enteros a partir de los datos proporcionados por el usuario. Los datos de la matriz se pedirán con el procedimiento de lectura de vectores, uno por fila, desechando los elementos que se escriban de más y rellenando con ceros los que se escriban de menos. Una vez construida la matriz, el programa ha de escribir sus elementos en pantalla, una fila por línea.

EJ5.36-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es nula. (Todos sus elementos iguales a cero)

EJ5.37-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es positiva. (Todos sus elementos mayores o iguales a cero)

EJ5.38-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es negativa. (Todos sus elementos menores o iguales a cero)

EJ5.39-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es diagonal. (Todos los elementos que no están en la diagonal principal son nulos).

EJ5.40-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es triangular superior. (Todos los elementos que están por debajo de la diagonal principal son nulos).

EJ5.41-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es triangular inferior. (Todos los elementos que están por encima de la diagonal principal son nulos).

EJ5.42-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es dispersa. (Todas las filas y todas las columnas contienen al menos un elemento nulo).

EJ5.43-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y compruebe si la matriz es simétrica. (Los elementos de la matriz (i, j) y (j, i), si existen, son iguales).

EJ5.44-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y construya un vector con la suma de todas las filas de la matriz inicial.

EJ5.45-ARR-FH

Desarrollar un programa en el que se pida al usuario una matriz de dimensiones N×M, y construya un vector con la suma de todas las columnas de la matriz inicial.

EJ5.46-ARR-FH

Desarrollar un programa en el que se pidan al usuario dos matrices de dimensiones N×M, y construya una nueva matriz representando la suma de las matrices iniciales.

EJ5.47-ARR-FH

Desarrollar un programa en el que se pidan al usuario dos matrices de dimensiones N×M, y construya una nueva matriz representando la resta de las matrices iniciales.

Tema 6: POO con Java

EJ6.1-POO-OP

Implementar una clase *Punto* cuyos datos miembro sean las coordenadas de un punto del plano. Estos datos han de ser privados. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Punto()* que recibe como argumentos dos números reales, *a* y *b* y construye un nuevo objeto de la clase *Punto* cuyas coordenadas son *a* y *b.*
2. Los métodos de acceso *getX()* y *getY()*, sin argumentos, que devuelven las coordenadas de un objeto *Punto*.
3. Los métodos modificadores *setX()* y *setY()*, que reciben un argumento y modifican el valor de la correspondiente coordenada de un objeto *Punto*.
4. El método *igual()*, que comprueba si un objeto de la clase *Punto* es igual a otro dado que se pasa como argumento.
5. El método *distancia()*, sin argumentos, que calcula la distancia de un objeto de la clase *Punto* al origen de coordenadas.
6. El método *distancia()*, que calcula la distancia de un objeto de la clase *Punto* a otro que se proporciona como argumento.

EJ6.2-POO-OP

Nota: este ejercicio depende del anterior.
Implementar una clase *Vector* para representar vectores. Los datos miembro de esta clase son las coordenadas del vector. Estos datos han de ser privados. Para esta clase se piden los siguientes constructores y métodos:

2. El constructor *Vector()* que recibe como argumentos dos números reales, *a* y *b* y construye un nuevo objeto de la clase *Vector* cuyas coordenadas *a* y *b*.
3. El constructor *Vector()* que recibe como argumento un objeto de la clase *Punto* y construye un nuevo objeto de la clase *Vector* cuyas coordenadas coinciden con las del objeto de la clase *Punto*.
4. El constructor *Vector()* que recibe como argumentos dos objetos de la clase *Punto*, *P1* y *P2*, y construye un nuevo objeto de la clase *Vector* que representa el vector de origen *P1* y extremo *P2*.
5. Los métodos de acceso *getX()* y *getY(*), sin argumentos, que devuelven las coordenadas de un objeto *Vector*.
6. Los métodos modificadores *setX()* y *setY()*, que reciben un argumento y modifican el valor de la correspondiente coordenada de un objeto *Vector*.
7. El método *igual()*, que comprueba si un objeto de la clase *Vector* es igual a otro dado que se pasa como argumento.
8. El método *longitud()*, sin argumentos, que calcula la longitud de un objeto de la clase *Vector*.
9. El método *proporcional()*, que comprueba si un objeto de la clase *Vector* es proporcional a otro dado que se pasa como argumento.

10. El método *perpendicular()*, que comprueba si un objeto de la clase *Vector* es perpendicular a otro dado que se pasa como argumento.
11. El método *traslada()*, que recibe como argumento un objeto de la clase *Punto*, P, y devuelve el objeto *Punto* resultado de trasladar el punto *P* usando un objeto de la clase *Vector*.

EJ6.3-POO-OP

Nota: este ejercicio depende del anterior.
Implementar una clase *Recta* para representar líneas rectas. Los datos miembro de esta clase son un objeto de la clase *Punto* perteneciente a la recta y un objeto de la clase *Vector* que representa la dirección de la recta. Estos datos han de ser privados. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Recta()* que recibe como argumentos un objeto de la clase *Punto*, P, y un objeto de la clase *Vector*, v, y construye un nuevo objeto de la clase *Recta* que representa a la recta que pasa por *P* con dirección *v*.
2. Los métodos de acceso *getPunto()* y *getVector()*, sin argumentos, que devuelven respectivamente los objetos *Punto* y *Vector*, datos miembro de un objeto *Recta*.
3. Los métodos modificadores *setPunto()* y setVector*()*, que respectivamente reciben como argumento un objeto de la clase *Punto* y un objeto de la clase *Vector*, y modifican el correspondiente dato miembro de un objeto *Recta*.
4. El constructor *Recta()* que recibe como argumento un objeto de la clase *Vector*, v, y construye un nuevo objeto de la clase *Recta* que representa a la recta que pasa por el origen de coordenadas con dirección v.
5. El constructor *Recta()* que recibe como argumentos dos objetos de la clase *Punto*, *P1* y *P2*, y construye un nuevo objeto de la clase *Recta* que representa a la recta que pasa por *P1* y *P2*.
6. El método perpendicular, que comprueba si un objeto de la clase *Recta* es perpendicular a otro dado que se pasa como argumento.
7. El método *paralela()*, que comprueba si un objeto *Recta* es paralelo a otro que se pasa como argumento.
8. El método *pertenece()*, que recibe como argumento un objeto *Punto P*, y comprueba si *P* se encuentra en la recta representado por un objeto *Recta*.
9. El método *igual()*, que comprueba si un objeto de la clase *Recta* es igual a otro dado que se pasa como argumento.
10. El método *paralelaPunto()*, que recibe como argumento un objeto *Punto P*, y construye la representación de la recta paralela a un objeto *Recta* que pasa por el punto *P*.
11. El método *perpendicularPunto()*, que recibe como argumento un objeto *Punto P*, y construye la representación de la recta perpendicular a un objeto *Recta* que pasa por el punto *P*.

EJ6.4-POO-OP

Nota: este ejercicio depende del anterior.

Implementar una clase *Segmento* para representar segmentos. Los datos miembro de esta clase son dos objetos de la clase *Punto* extremos de un segmento. Estos datos han de ser privados. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Segmento()* que recibe como argumentos dos objetos de la clase *Punto*, *P1* y *P2*, y construye un nuevo objeto de la clase *Segmento* cuyos extremos son *P1* y *P2*.
2. Los métodos de acceso *getA()* y *getB()*, sin argumentos, que devuelven los extremos de un objeto *Segmento*.
3. Los métodos modificadores *setA()* y *setB()*, que reciben como argumento un objeto de la clase *Punto* y modifican el correspondiente extremo de un objeto *Segmento*.
4. El método *igual()*, que comprueba si un objeto de la clase *Segmento* es igual a otro dado que se pasa como argumento.
5. El método *longitud()*, sin argumentos, que calcula la longitud de un objeto de la clase *Segmento*.
6. El método *proporcional()*, que comprueba si un objeto de la clase *Segmento* es proporcional a otro dado que se pasa como argumento.
7. El método *perpendicular()*, que comprueba si un objeto de la clase *Segmento* es perpendicular a otro dado que se pasa como argumento.
8. El método *puntoMedio()*, sin argumentos, que devuelve el objeto *Punto* que representa el punto medio de un objeto *Segmento*.
9. El método *pertenece()*, que recibe como argumento un objeto *Punto* P, y comprueba si P se encuentra en el segmento representado por un objeto *Segmento*.
10. El método *recta()*, sin argumentos, que construye un objeto *Recta* que contiene a un objeto *Segmento*.
11. El método *mediatriz()*, sin argumentos, que devuelve un objeto *Recta* que representa la mediatriz de un objeto *Segmento*.

EJ6.5-POO-OB

Diseñe una clase *Cuenta* que represente una cuenta bancaria y permita realizar operaciones como ingresar y retirar una cantidad de dinero, así como realizar una transferencia de una cuenta a otra. Se pide:

1. Represente gráficamente la clase.
2. Defina la clase utilizando la sintaxis de Java, definiendo las variables de instancia y métodos que crea necesarios.
3. Implemente cada uno de los métodos de la clase. Los métodos deben actualizar el estado de las variables de instancia y mostrar un mensaje en el que se indique que la operación se ha realizado con éxito.
4. Cree un programa en Java (en una clase llamada *CuentaTest*) que cree un par de objetos de tipo *Cuenta* y realice operaciones con ellos. El programa debe comprobar que todos los métodos de la clase *Cuenta* funcionan correctamente.

EJ6.6-HER-OBE

¿Cuál es el resultado del siguiente programa?

```
1  class Uno {
2          protected int i=2;
3          public void frase() {
4                  System.out.println("Estoy en un objeto de clase Uno");
5          }
6  }
```

```
1  class Dos extends Uno {
2          public void frase() {
3                  int i=3;
4                  System.out.println("Estoy en un objeto de clase Dos con i:"+i);
5          }
6  }
```

```
1  class Tres extends Dos {
2          public void frase() {
3                  System.out.println("Estoy en un objeto de clase Tres con i:"+i);
4          }
5  }
6
```

```
1  class Driver {
2          public static void main(String[] args) {
3                  Uno [] lista =new Uno [2];
4                  lista [0]= new Dos();
5                  lista [1]= new Tres();
6                  for (int i=0; i<2; i++){
7                          lista[i].frase();
8                  }
9          }
10 }
```

EJ6.7-HER-OBE

¿Cuál es el resultado del siguiente programa?

```
1  class Padre {
2          protected int aa=0;
3          public int aa() {
4                  return aa;
5          }
6  }
```

```
1  class Hija extends Padre {
2          public Hija (int bb) {
3                  this.aa=bb+1;
```

```
4           }
5   }
```

```
1   class Nieta extends Hija {
2           public Nieta (int cc) {
3                   super(cc+2);
4           }
5   }
```

```
1   class Familia {
2           private static Nieta f (Padre h) {
3                   Nieta n=new Nieta (h.aa());
4                   return n;
5           }
6           public static void main(String [] args){
7                   Hija h= new Hija(4);
8                   h=f(h);
9                   System.out.println (h.aa());
10          }
11  }
```

EJ6.8-HER-OB

En una tienda se venden 2 tipos de ordenadores: portátiles y de sobremesa. Ambos tipos de ordenadores se caracterizan por su código y por su precio. Además cada uno tiene un eslogan que es: "*Ideal para sus viajes*" en el caso de los portátiles y "*Es el que más pesa, pero el que menos cuesta*" para el caso de los ordenadores de sobremesa. Además los ordenadores portátiles tienen un atributo peso, y los de sobremesa la descripción del tipo de torre.

1. Represente gráficamente las clases que considere necesarias, así como su relación.
2. Implemente en Java dichas clases.

EJ6.9-HER-OB

Desarrollar una clase *Empresa* cuyos datos miembro sean un nombre, un tamaño y un array de empleados personal (la clase *Empleado* se pide en el siguiente ejercicio). El tamaño de la empresa es inmutable. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Empresa()* que recibe como argumentos una cadena de texto *nombre* y un valor entero *tamaño*, y construye un nuevo objeto de la clase *Empresa* cuyo nombre y tamaño vienen dados respectivamente por los argumentos *nombre* y *tamaño*. El tamaño del array de empleados personal viene dado por el valor de la variable *tamaño*.
2. Los métodos de acceso *getNombre()* y *getTamaño()*, sin argumentos, que respectivamente devuelven el nombre y el tamaño de un objeto *Empresa*.

3. El método de acceso *getEmpleado()*, que recibe como argumento un número entero menor que el tamaño de la empresa, y devuelve el correspondiente campo del array de empleados.
4. El método *despideEmpleado()*, que recibe como argumento un número entero menor que el tamaño de la empresa, y asigna la referencia null al correspondiente campo del array de empleados.

EJ6.10-HER-OB

Nota: este ejercicio depende del anterior.
Desarrollar una clase *Empleado* cuyos datos miembro sean una empresa, un *nombre*, un *sueldo* y un número de empleado (*numEmpleado*). Estos datos han de ser protegidos (*protected*). Además, el número de empleado y la empresa son inmutables. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Empleado()* que recibe como argumentos *emp*, una referencia a un objeto *Empresa*, una cadena de texto *nombre* y un valor entero *sueldo*, y construye un nuevo objeto de la clase *Empleado* en el que la empresa, el nombre y el sueldo vienen dados respectivamente por los argumentos *emp*, *nombre* y *sueldo*. El número de empleado se crea de manera única por cada empleado, usando para ello una variable contador perteneciente a la clase (*static* y *private*).
2. El constructor protegido *Empleado()* que recibe como argumentos *emp*, una referencia a un objeto *Empresa*, una cadena de texto no*()*mbre, un valor entero *sueldo* y un número de empleado *numero*, y construye un nuevo objeto de la clase *Empleado* en el que la empresa, el nombre, el sueldo y el número de empleado vienen dados respectivamente por los argumentos *emp*, *nombre*, *sueldo* y *numero*.
3. Los métodos de acceso *getNombre()*, *getSueldo()* y *getNumeroEmpleado()*, sin argumentos, que respectivamente devuelven los campos *nombre*, *sueldo* y *numEmpleado* de un objeto *Empleado*.
4. El método modificador *setNombre()*, que recibe como argumento una cadena y modifica el valor del campo *nombre* de un objeto *Empleado*.
5. El método modificador *setSueldo()*, que recibe como argumento un valor entero y modifica el valor del campo *sueldo* de un objeto *Empleado*.
6. El método de impresión en pantalla *toString()*, sin argumentos, que devuelve una cadena (String) con el nombre, número y sueldo de un objeto *Empleado*.
7. El método *aumentarSueldo()*, que recibe como argumento un número entero N y modifica el sueldo del objeto *Empleado* sobre el que se evalúa, aumentándolo un N%. Este método no puede ser modificado por clases derivadas.
8. El método *despedir()*, sin argumentos, que despide al objeto *Empleado* sobre el que se evalúa.

EJ6.11-HER-OB

Nota: este ejercicio depende del anterior.
Añadir a la clase *Empresa* el método *nuevoEmpleado()*, que recibe como argumentos una cadena de texto *nombre* y un valor entero *sueldo*, crea un nuevo empleado asignado a la empresa sobre la que se evalúa el método, cuyo nombre y sueldo vienen

dados por los argumentos nombre y sueldo y, finalmente, utiliza el número de empleado como índice para almacenar una referencia al objeto *Empleado* recién creado en el array de empleados de la empresa.

EJ6.12-HER-OB

Nota: este ejercicio depende del anterior.
Desarrollar una clase *Ejecutivo* derivada de la clase *Empleado* anterior, con un campo entero adicional presupuesto. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Ejecutivo()* que recibe como argumentos *emp*, una referencia a un objeto *Empresa*, una cadena de texto *nombre* y un valor entero *sueldo*, y construye un nuevo objeto de la clase *Ejecutivo* utilizando el constructor de la clase base (*super*(...)).
2. El constructor *Ejecutivo()* que recibe como argumentos *emp*, una referencia a un objeto *Empresa*, una cadena de texto *nombre*, un valor entero *sueldo* y un número de empleado *numero*, y construye un nuevo objeto de la clase *Ejecutivo* utilizando el constructor de la clase base (*super*(...)).
3. El método de acceso *getPresupuesto()*, sin argumentos, que devuelve el valor del campo presupuesto de un objeto *Ejecutivo*.
4. El método modificador *asignaPresupuesto()*, que recibe como argumento un valor entero y modifica el valor del campo presupuesto de un objeto *Ejecutivo*.
5. Redefinir el método de impresión en pantalla *toString()* para que indique que el objeto sobre el que se evalúa es un ejecutivo.
6. Añadir a la clase *Empleado* el método *ascender()* sin argumentos, que crea un nuevo objeto *Ejecutivo* con los datos del objeto *Empleado* sobre el que se evalúa el método y cambia la referencia en el array de personal de la empresa a la que pertenece dicho objeto.

EJ6.13-HER-OP

Desarrollar una clase *Producto* cuyos datos miembro sean una cadena de texto identificación y un valor real (double) *precioBase*. Ambos datos de tipo protegido. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Producto()* que recibe como argumentos una cadena de texto identificación y un valor real (double) *precioBase*, y construye un nuevo objeto de la clase *Producto* cuya identificación y precio base vienen dados respectivamente por los argumentos *identificación* y *precioBase*.
2. Los métodos de acceso *getIdentificacion()* y *getPrecioBase()*, sin argumentos, que respectivamente devuelven el *identificador* y el *precioBase* de un objeto *Producto*.
3. El método modificador *setIdentificacion()*, que recibe como argumento una cadena de texto identificación y cambia el campo identificación del objeto sobre el que se aplica, asignándole como nuevo valor el del argumento *identificación*.

4. El método modificador *setPrecioBase()*, que recibe como argumento un valor real (double) *precioBase* y cambia el campo *precioBase* del objeto sobre el que se aplica, asignándole como nuevo valor el del argumento *precioBase*.
5. El método de impresión en pantalla *toString()*, sin argumentos, tal que al ser evaluado sobre un objeto de tipo *Producto* cuya identificación es "RJ45" y cuyo precio base es 10.50, genere la siguiente cadena: RJ45 (10.5)

EJ6.14-HER-OP

Nota: este ejercicio depende del anterior.
Desarrollar una clase *ProductoInventariado* derivada de la clase *Producto* anterior, con dos campos enteros adicionales *cantidad* y *beneficio*. Ambos de tipo protegido. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *ProductoInventariado()*, que recibe como argumentos una cadena de texto identificación, un valor real (double) *precioBase* y dos datos enteros cantidad y beneficio, y construye un nuevo objeto de la clase *Producto* utilizando el constructor de la clase base (*super*(...)), cuya identificación, precio base, cantidad y beneficio vienen dados respectivamente por los argumentos *identificación*, *precioBase*, *cantidad* y *beneficio*.
2. Los métodos de acceso *getCantidad()* y *getBeneficio()*, sin argumentos, que respectivamente devuelven la cantidad y el beneficio de un objeto *ProductoInventariado*.
3. Los métodos modificadores *setCantidad()* y *setBeneficio()*, que reciben como argumento un valor entero y modifican, respectivamente, el valor del campo *cantidad* y *beneficio* del objeto sobre el que se aplican, asignándoles como nuevo valor el del argumento recibido.
4. El método *precioFinal()*, sin argumentos, que devuelve el precio final de un objeto de la clase *ProductoInventariado* determinado como el precio base al que se suma el porcentaje de beneficio.
5. El método de impresión en pantalla *toString()* sin argumentos, tal que al ser evaluado sobre un objeto de tipo *ProductoInventariado* cuya identificación es "RJ45", cuyo precio base es 10.50, cuya cantidad es 13 y cuyo beneficio es 10, genera la siguiente cadena: 10 RJ45 (10.5) (+13%)

EJ6.15-HER-OP

Nota: este ejercicio depende del anterior.
Desarrollar una clase *Tienda* cuyos datos miembro son una cadena de texto nombre, un valor entero inmutable *maxProducto* que indica el número máximo de productos distintos que puede tener la tienda, inventario un array de objetos del tipo *ProductoInventariado* donde se almacena información de los productos de la tienda, un valor entero *ultimaEntrada* que indica la primera posición libre en el array inventario y un valor real (double) caja que almacena el dinero del que dispone la tienda. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Tienda()*, que recibe como argumentos una cadena de texto *nombre*, un valor entero *maxProducto* y un valor real *caja*, y construye un nuevo objeto de la clase *Tienda* cuyo *nombre*, *maxProducto* y *caja* vienen dados respectivamente por los argumentos *nombre*, *maxProducto* y *caja*. El tamaño del array *inventario* viene dado por el valor del argumento *maxProducto* y el valor de *ultimaEntrada* es 0.3
2. El método *buscaProducto()*, que recibe como argumento una cadena de texto *id*, y devuelve el índice del array *inventario* en el que se encuentra un producto cuyo identificador coincide con *id*, si es que existe, o el valor de *ultimaEntrada* en caso contrario.
3. El método *añadirProducto()*, que recibe como argumento un *identificador* de producto *id*, un precio base *p* (tipo double), una cantidad *c* (tipo int) y un beneficio *b* (tipo int) y lo añade al *inventario*. Si el producto ya estaba en el *inventario* entonces solo hay que modificar los datos relativos al precio base, cantidad y beneficio. Si el producto no está en el *inventario* entonces hay que añadirlo. En cualquier caso, solo se podrá añadir un producto si el coste total (cantidad × precio base) es menor o igual que el dinero del que dispone la tienda, el cual ha de ser disminuido de manera adecuada. Si en el inventario no hay sitio para el producto o éste no puede ser adquirido por no disponer de suficiente dinero entonces se ha de indicar en pantalla un mensaje informativo.
4. El método *venderProducto()*, que recibe como argumento un identificador de producto *id* y una cantidad *c* y, si el producto existe en el inventario en una cantidad mayor o igual que *c*, entonces disminuye en *c* unidades la cantidad del producto *id* que hay en el *inventario* y modifica adecuadamente la *caja*. Si la tienda se queda sin unidades del producto *id* entonces hay que modificar adecuadamente el array *inventario* y el valor de *ultimaEntrada* para evitar "huecos vacíos". Si no hay unidades suficientes del producto *id* para vender, entonces se ha de indicar en pantalla un mensaje informativo.

EJ6.16-INT-OB

Implemente el código de una interfaz llamada *Primera* que contenga dos métodos *A()* y *B()*. Defina otra interfaz llamada Segunda que herede de la anterior y además contenga un método llamado *C()*. Escriba el código de otra clase llamada *Objetos* que use la segunda interfaz. ¿Cuántos métodos debe implementar esta clase? Implemente dichos métodos de forma que cada método imprima una línea indicando el nombre del método. Cree un programa que utilice los métodos definidos.

EJ6.17-INT-OBE

¿Cuál es el error del siguiente programa?

```
1  interface A {
2          double a=2.3;
3          void imprimeresultadoA () {
4                  System.out.println ("valor de a"+ a);
5          }
6  }
```

```
1  interface B {
2          int b=435;
3          void imprimeresultadoB () ;
4  }
```

```
1  class AA implements A, B {
2          double aa=324.32;
3          public void imprimeresultadoA () {
4                  System.out.println ("valor de a"+ a+ "valor de aa"+ aa);
5          }
6          public void imprimeresultadoB () {
7                  System.out.println ("valor de b"+ b+ "valor de aa"+ aa);
8          }
9  }
```

```
1  class Principal {
2          public static void main (String [] args) {
3                  AA ob1=new AA();
4                  ob1.imprimeresultadoA();
5                  ob1.imprimeresultadoB();
6          }
7  }
```

EJ6.18-INT-OP

Desarrollar una clase *Parking*, cuyos datos miembro sean un nombre, un entero *plazasTotal*, indicando el número total de plazas, un entero *plazasOcupadas*, indicando el número total de plazas ocupadas, un entero *plazasAbonados*, indicando el número total de plazas reservadas para clientes abonados, un entero *plazasHotel*, indicando el número total de plazas reservadas para clientes del hotel y un array plazas de *Vehículos*. El campo *plazasTotal* es inmutable. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *Parking()* que recibe como argumentos una cadena de texto *nombre* y un valor entero *plazasTotal*, y construye un nuevo objeto de la clase *Parking* cuyo nombre y número total de plazas vienen dados respectivamente por los argumentos *nombre* y *plazasTotal*. El tamaño del array plazas de vehículos viene dado por el valor de la variable *plazasTotal*.
2. Los métodos de acceso *getNombre()*, *getPlazasTotal()*, *getPlazasOcupadas()*, *getPlazasAbonados()*, *getPlazasHotel()* y *getPlazasClientes()*, sin argumentos, que devuelven, respectivamente, el nombre, el número total de plazas, el número de plazas ocupadas, el número de plazas reservadas a clientes abonados, el número de plazas reservadas a clientes del hotel y el número de plazas restantes dedicadas a clientes externos.
3. El método de acceso *getVehiculo()*, que recibe como argumento un número entero, y, si dicho número es menor que el número total de plazas del parking

entonces devuelve el correspondiente campo del array de plazas. En caso contrario devuelve una referencia vacía (*null*).

4. El método de modificación *setPlazasAbonados()*, que reciben como argumento un número entero y, si se dispone de dicho número de plazas en el parking, se almacena dicho valor en el campo *plazasAbonados*. Hay que tener en cuenta las plazas que ya están reservadas para los vehículos del hotel.
5. El método de modificación *setPlazasHotel()*, que reciben como argumento un número entero y, si se dispone de dicho número de plazas en el parking, se almacena dicho valor en el campo *plazasHotel*. Hay que tener en cuenta las plazas que ya están reservadas para los vehículos de clientes abonados.
6. El método *numAbonados()*, sin argumentos, que devuelve el número de plazas del parking ocupadas por vehículos de abonados. La clase que almacena información sobre los vehículos de los abonados es *VehiculoAbonado*.
7. El método *numHotel()*, sin argumentos, que devuelve el número de plazas del parking ocupadas por vehículos del hotel. La clase que almacena información sobre los vehículos del hotel es *VehiculoHotel*.
8. El método *numClientes()*, sin argumentos, que devuelve el número de plazas del parking ocupadas por vehículos del hotel. La clase que almacena información sobre los vehículos del hotel es *VehiculoCliente*.
9. El método de modificación *setVehiculo()*, que recibe como argumento una referencia a un objeto de tipo *Vehiculo* y, si hay sitio en el parking, sitúa dicha referencia en una plaza libre. En caso de no existir plazas libres se tiene que presentar un mensaje informativo en pantalla.

EJ6.19-INT-OP

Desarrollar una interfaz *Vehiculo* que declare los métodos *factura()*, *hayPlaza()*, *aparca()* y setTiempo*()*, tales que: *factura* (sin argumentos) proporciona el importe a pagar por estacionar un *Vehiculo* durante determinado tiempo en un parking. *hayPlaza()* recibe como argumento una referencia a un objeto *Parking* y determina si hay sitio en dicho *Parking* para estacionar un *Vehiculo*. *aparca()* recibe como argumento una referencia a un objeto *Parking* y estaciona un *Vehiculo* en dicho *Parking*. *setTiempo()* recibe como argumento un entero y establece ese entero como tiempo de estancia del vehículo en el parking.

EJ6.20-INT-OP

Desarrollar una clase *VehiculoCliente* implementando la interfaz *Vehiculo*, cuyos datos miembro son una cadena de texto id y un valor entero tiempo. La cadena de texto id funciona como identificación del vehículo y no puede ser modificada. El valor entero tiempo almacena el número de horas que el vehículo está en el parking. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *VehiculoCliente()*, que recibe como argumento un identificador de vehículo, y construye un nuevo objeto del tipo *VehiculoCliente*. El valor inicial del campo tiempo es 0.

2. El método *setTiempo()*, que recibe como argumento un número entero y lo almacena como valor del campo tiempo del objeto *VehiculoCliente*.
3. El método *factura()*, sin argumentos, que determina lo que tiene que pagar un objeto del tipo *VehiculoCliente* por su estancia en el parking. El coste de la estancia es de 12€ el día completo y 0.6€ la hora o fracción.
4. El método *hayPlaza()*, que recibe como argumento una referencia a un objeto *Parking* y determina si hay una plaza en dicho *Parking* para estacionar un objeto del tipo *VehiculoCliente*.
5. El método *aparca()*, que recibe como argumento una referencia a un objeto *Parking* y estaciona un *VehiculoCliente* en dicho *Parking*.

EJ6.21-INT-OP

Desarrollar una clase *VehiculoAbonado* implementando la interfaz *Vehiculo*, cuyos datos miembro son una cadena de texto *id* y un valor entero *tiempo*. La cadena de texto *id* funciona como identificación del vehículo y no podrá ser modificada. El valor entero tiempo almacena el número de meses para los que el vehículo tiene contratado el parking. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *VehiculoAbonado()*, que recibe como argumento un identificador de vehículo y un número entero tiempo indicando el número de meses para los que se ha contratado el parking, y construye un nuevo objeto del tipo *VehiculoAbonado*.
2. El método *factura()*, sin argumentos, que determina lo que tiene que pagar un objeto del tipo *VehiculoAbonado* por su estancia en el parking. El coste de la estancia es de 200€ al mes.
3. El método *hayPlaza()*, que recibe como argumento una referencia a un objeto *Parking* y determina si hay una plaza en dicho parking para estacionar un objeto del tipo *VehiculoAbonado*.
4. El método *aparca()*, que recibe como argumento una referencia a un objeto Parking y estaciona un *VehiculoAbonado* en dicho parking.

EJ6.22-INT-OP

Desarrollar una clase *VehiculoHotel* implementando la interfaz *Vehiculo*, cuyos datos miembro son una cadena de texto *id* y un valor entero *tiempo*. La cadena de texto *id* funciona como identificación del vehículo y es inmutable. El valor entero tiempo almacena el número de días para los que el vehículo tiene concertado el parking. Para esta clase se piden los siguientes constructores y métodos:

1. El constructor *VehiculoHote()*, que recibe como argumento un identificador de vehículo y un número entero tiempo indicando el número de días para los que se ha concertado el parking, y construye un nuevo objeto del tipo *VehiculoHotel*.
2. El método *factura()*, sin argumentos, que determina lo que tiene que pagar un objeto del tipo *VehiculoHotel* por su estancia en el parking. El coste de la estancia es de 10€ al día.

3. El método *hayPlaza()*, que recibe como argumento una referencia a un objeto Parking y determina si hay una plaza en dicho parking para estacionar un objeto del tipo VehiculoHotel.
4. El método *aparca()*, que recibe como argumento una referencia a un objeto parking y estaciona un *VehiculoHotel* en dicho parking.

EJ6.23-INT-OP

Una centralita gestiona las llamadas telefónicas de un conjunto de clientes. Cada cliente está identificado de manera única en la centralita y dispone de un saldo. Para cada llamada se guarda el número de teléfono al que se llama y la duración de la misma. La centralita guarda un registro de las últimas llamadas realizadas por los clientes. Las llamadas pueden ser locales o provinciales. El coste de una llamada local es de 0.15€ por segundo. El coste de una llamada provincial depende de la franja horaria en la que se realiza dicha llamada: 0.20€ en franja 1, 0.25€ en franja 2 y 0.30€ en franja 3.

1. Implementar la interfaz Llamada y las clases *LlamadaLocal* y *LlamadaProvincial*, para almacenar la información relativa a las llamadas. En estas clases se ha de poder calcular el coste de la llamada.
2. Implementar la clase *Cliente*, para almacenar la información relativa a los clientes de la centralita.
3. Implementar la clase *Registro*, para almacenar la información de cada uno de los registros que almacena la centralita.
4. Implementar la clase *Centralita* de forma que almacene información sobre los clientes y el registro de llamadas de estos. Las operaciones que se han de poder hacer en la centralita son las siguientes: Dar de alta a un cliente. Dar de baja a un cliente. Incrementar el saldo de un cliente (o darlo de alta si es que no existe). Añadir un nuevo registro al registro de llamadas.
5. Presentar en pantalla la lista de los últimos N registros (0 <= N <= 100).

Tema 7: Colecciones y Tipos Genéricos

EJ7.1-COL-OB

Recorrido de vectores y listas

```
1  import java.util.*;
2  public class Ejercicio {
3          public static void main(String[] args) {
4                  // Creamos un Vector con 10 cadenas
5                  Vector v = new Vector();
6                  for (int i = 0; i < 10; i++)
7                          v.addElement("Hola" + i);
8                  // Recorrido mediante Enumeration
9                  // Recorrido mediante Iterator
10                 // Recorrido mediante un bucle for, accediendo a mano a cada posición
   del vector
11         }
12 }
```

La clase anterior tiene un método *main*, donde hemos creado un objeto de tipo *Vector*, y le hemos añadido 10 cadenas: *Hola0, Hola1, Hola2...Hola9*. Con este vector deberás hacer lo siguiente:

1. Primero, recorrerlo mediante el objeto *Enumeration* que puedes obtener del propio vector. Si observas la API de Vector, verás que tiene un método:
   Enumeration elements();
   Devuelve un *Enumeration* para poder recorrer los elementos del vector. Se trata de que obtengas esa enumeración, y formes un bucle para recorrerla de principio a fin. Para cada elemento, saca su valor por pantalla (imprime la cadena).
2. A continuación de lo anterior, haz otro recorrido del vector, pero esta vez utilizando su Iterator. Verás también en la API que el objeto Vector tiene un método:
   Iterator iterator();
   Devuelve un *Iterator* para poder recorrer los elementos del vector. Haz ahora otro bucle como el que se explica en los apuntes, para recorrer los elementos del vector, esta vez con el *Iterator*. Para cada elemento, vuelve a imprimir su valor por pantalla.
3. Finalmente, tras los dos bucles anteriores, añade un tercer bucle, donde "a mano" vayas recorriendo todo el vector, accediendo a sus elementos, y sacándolos por pantalla. En este caso, ya no podrás utilizar los métodos *nextElement()*, *hasNext()*, ni similares que has utilizado en los bucles anteriores. Deberás ir posición por posición, accediendo al valor de esa posición del vector, y sacando el valor obtenido por pantalla.
4. Una vez tengas los tres bucles hechos, ejecuta el programa, y observa lo que saca cada uno de los bucles por pantalla. ¿Encuentras alguna diferencia en el

comportamiento de cada uno? ¿Qué forma de recorrer el vector te resulta más cómoda de programar y por qué?

NOTA: algunas de las técnicas que has utilizado para recorrer el vector se pueden utilizar de la misma forma para recorrer otros tipos de listas. Por ejemplo, puedes obtener el *Iterator* de un *ArrayList* y recorrerlo, o ir elemento por elemento.

EJ7.2-COL-OP

Pruebas de eficiencia

```
1  import java.util.*;
2  public class Ejercicio {
3          public static void main(String[] args) {
4                  System.out.println ("CreaYBorraEnmedio:");
5                  creaYBorraEnmedio(10000);
6                  System.out.println ("CreaYBorraFinal:");
7                  creaYBorraFinal(10000000);
8          }
9
10         public static void creaYBorraEnmedio(int N) {
11                 // Creamos un Vector con 10000 cadenas
12                 Vector v = new Vector();
13                 for (int i = 0; i < 10000; i++)
14                         v.addElement("Hola" + i);
15
16                 // Creamos una LinkedList con 10000 cadenas
17                 LinkedList ll = new LinkedList();
18                 for (int i = 0; i < 10000; i++)
19                         ll.addLast("Hola" + i);
20
21                 //Hacemos "cantidad" operaciones de inserción y "cantidad" de borrado en el medio del vector
22                 // y anotamos el tiempo
23                 long t1 = System.currentTimeMillis();
24                 for (int i = 0; i < N; i++) {
25                         v.add(v.size()/2, "Hola" + i);
26                         v.remove(v.size()/2);
27                 }
28                 long t2 = System.currentTimeMillis();
29                 System.out.println("Tiempo con vector (enmedio): " + (t2 - t1) + " ms");
30
31                 // Hacemos lo mismo con la LinkedList
32                 t1 = System.currentTimeMillis();
33                 for (int i = 0; i < N; i++) {
34                         ll.add(ll.size()/2, "Hola" + i);
35                         ll.remove(ll.size()/2);
36                 }
37                 t2 = System.currentTimeMillis();
38                 System.out.println("Tiempo con LinkedList (enmedio): " + (t2 - t1) + " ms");
39         }
40
```

```
41          public static void creaYBorraFinal(int N) {
42                  // Creamos un Vector con 10000 cadenas
43                  Vector v = new Vector();
44                  for (int i = 0; i < 10000; i++)
45                          v.addElement("Hola" + i);
46
47                  // Creamos una LinkedList con 10000 cadenas
48                  LinkedList ll = new LinkedList();
49                  for (int i = 0; i < 10000; i++)
50                          ll.addLast("Hola" + i);
51
52                  //Hacemos "cantidad" operaciones de inserción y "cantidad" de borrado al final del vector
53                  // y anotamos el tiempo
54                  long t1 = System.currentTimeMillis();
55                  for (int i = 0; i < N; i++) {
56                          v.add(v.size()-1, "Hola" + i);
57                          v.remove(v.size()-1);
58                  }
59                  long t2 = System.currentTimeMillis();
60                  System.out.println("Tiempo con vector (final): " + (t2 - t1) + " ms");
61
62                  // Hacemos lo mismo con la LinkedList
63                  t1 = System.currentTimeMillis();
64                  for (int i = 0; i < N; i++)  {
65                          ll.addLast("Hola" + i);
66                          ll.removeLast();
67                  }
68                  t2 = System.currentTimeMillis();
69                  System.out.println("Tiempo con LinkedList (final): " + (t2 - t1) + " ms");
70          }
71  }
```

La clase anterior contiene un método *main* que a su vez llama a dos métodos de la propia clase:

El método *creaYBorraEnmedio()* crea un Vector de 10.000 cadenas, y una *LinkedList* con otras 10.000 cadenas. Después, hace N inserciones y borrados en la parte media del Vector y del *LinkedList*, y compara los tiempos que se ha tardado en uno y otro tipo de datos en hacer todas las operaciones.
El método *creaYBorraFinal()* crea un Vector de 10.000 cadenas, y una *LinkedList* con otras 10.000 cadenas. Después, hace N inserciones y borrados en la parte final del *Vector* y del *LinkedList*, y compara los tiempos que se ha tardado en uno y otro tipo de datos en hacer todas las operaciones.

El método *main* prueba el primer método con N = 10.000 operaciones, y el segundo con N = 1.000.000 operaciones. Se pide:

1. Ejecuta el programa y observa los tiempos de ejecución de *creaYBorraEnmedio()*. ¿Qué conclusiones sacas?

2. Observa también los tiempos de ejecución de *creaYBorraFinal()*. ¿Qué conclusiones sacas en este otro caso?
3. A la vista de los resultados... ¿en qué casos crees que es mejor utilizar *LinkedList*, y en qué otros no es aconsejable hacerlo?
4. Finalmente, añade un tercer método *creaYBorraInicio()* que haga lo mismo que los anteriores, pero haciendo las N inserciones y borrados por el inicio del *Vector* y de la *LinkedList* (para ésta, utiliza los métodos *addFirst()* y *removeFirst()*). Para este caso, haz que N sea de 1.000.000. ¿Qué conclusiones obtienes al ejecutar este tercer método?

EJ7.3-COL-OB

Trabajar con conjuntos

```
1  import java.util.*;
2  public class Ejercicio {
3          Vector v;
4          HashSet hs;
5
6          public Ejercicio () {
7                  v = new Vector();
8                  v.add("a1");
9                  v.add("a2");
10                 v.add("a3");
11
12                 hs = new HashSet();
13                 hs.add("a1");
14                 hs.add("a2");
15                 hs.add("a3");
16         }
17
18         public static void main(String[] args) {
19                 Ejercicio ejer = new Ejercicio();
20
21                 ejer.addVector(args[0]);
22                 ejer.addConjunto(args[0]);
23         }
24
25         public void addVector(String cadena) {
26         }
27
28         public void addConjunto(String cadena) {
29         }
30 }
```

Aunque tienen su utilidad, normalmente los tipos de conjuntos no se suelen emplear demasiado a la hora de programar. Su programación es muy similar a la que pueda tener un *ArrayList* o un *Vector*, y siempre se tiende a utilizar estos, porque sus clases son más conocidas. Sin embargo, los conjuntos tienen su verdadera utilidad cuando queremos tener listas de elementos no repetidos. Muchos programadores tienden a

hacer ellos "a mano" la comprobación de si está o no repetido, y con estas clases se facilitaría bastante la tarea.

La clase anterior recibe una cadena (sin espacios) como parámetro, y la añade dentro de sus campos *Vector* y *HashSet*. Dichos campos ya tienen insertadas las cadenas "*a1*", "*a2*" y "*a3*".

1. Rellena los métodos *addVector()* y *addConjunto()* para que añadan al vector o al conjunto, respectivamente, el elemento que se les pasa como parámetro, siempre que no exista ya. Tras insertarlo, deberán imprimir por pantalla su contenido actualizado (para recorrer el conjunto *HashSet* deberás acceder a su *Iterator*, probablemente).
2. Prueba a ejecutar el programa, pasándole como parámetro tanto una cadena que no exista ("*b1*", por ejemplo), como otra que sí ("*a2*", por ejemplo). Comprueba que en los dos casos se hace la inserción cuando toca, y se sacan bien los datos por pantalla. ¿Observas alguna diferencia en el orden en que se muestran los datos por pantalla cuando se hacen nuevas inserciones? ¿A qué crees que puede deberse?
3. Comenta las ventajas e inconvenientes que encuentres a la hora de programar con tipos conjunto como *HashSet* frente a tipos lista como *Vector*.

EJ7.4-COL-OP

Ventajas de los mapas

```
1  public class Ejercicio {
2          public static void main(String[] args) {
3                  ArrayList al = new ArrayList();
4
5                  // Añadir los elementos en la lista
6
7                  for (int i = 0; i < 10; i++) {
8                          al.add(new Parametro("Clave" + i, "Valor" + i));
9                  }
10
11                 // Buscar el elemento que se pasa como parametro
12
13                 int i = 0;
14                 boolean encontrado = false;
15                 while (!encontrado && i < al.size()) {
16                         Parametro p = (Parametro)(al.get(i));
17                         if (p.getNombre().equals(args[0]))
18                                 encontrado = true;
19                         else
20                                 ++i;
21                 }
22                 if (encontrado)
23                         System.out.println ("Finalizado. Encontrado en posicion " + i);
24                 else
25                         System.out.println ("Finalizado. No encontrado.");
```

```
26
27                      // Imprimir todos los elementos por pantalla, con su nombre y su valor
28
29                      for (i = 0; i < al.size(); i++) {
30                              Parametro p = (Parametro)(al.get(i));
31                              System.out.println (p.getNombre() + " = " + p.getValor());
32                      }
33              }
34  }
```

```
1   class Parametro {
2               String nombre;
3               String valor;
4
5               public Parametro (String n, String v) {
6                       nombre = n;
7                       valor = v;
8               }
9
10              public String getNombre() {
11                      return nombre;
12              }
13
14              public String getValor() {
15                      return valor;
16              }
17  }
```

Trabajar con mapas es la forma más eficiente y cómoda de almacenar pares clave-valor. La clase *Ejercicio* contiene una subclase llamada *Parametro*, que utilizamos para guardar ciertos parámetros de configuración, y sus valores. Verás que esta clase tiene un campo nombre donde pondremos el nombre del parámetro, y otro valor con su valor.

La clase principal *EjMapas* crea muchos parámetros de este tipo, y los almacena en un *ArrayList*. Finalmente, busca en dicho *ArrayList* el valor del parámetro cuya clave se le pasa en el *main*. Saca un mensaje indicando en qué posición lo encontró, y luego imprime todos los parámetros por pantalla, sacando su nombre y su valor.

1. Haz una clase *EjercicioHash* que haga lo mismo, pero utilizando una *Hashtable* en lugar de un *ArrayList*. Debes tener en cuenta lo siguiente:
   En esta clase no tendrás que usar la subclase *Parametro*, ya que podrás almacenar el nombre por un lado y el valor por el otro dentro de la tabla hash. Es decir, donde antes hacías:

   ArrayList al = new ArrayList();
   ...
   al.add(new Parametro("Clave1", "Valor1"));

ahora harás:

```
Hashtable ht = new Hashtable();
...
ht.put("Clave1", "Valor1");
```

A la hora de buscar el elemento en la hash ya no necesitas ningún bucle. El método *get()* de la *Hashtable* te permite obtener un valor si le das el nombre con que lo guardaste. Te devolverá el objeto asociado a ese nombre (como un *Object* que deberás convertir al tipo adecuado), o null si no lo encontró.

```
String valor = (String)(ht.get(nombre));
if (valor == null) ... else ...
```

En este caso no hace falta que indiques en qué posición encontraste al elemento, puesto que, como verás después, las tablas hash no mantienen las posiciones como te esperas.

A la hora de imprimir todos los elementos por pantalla, una opción es obtener todo el listado de claves, y luego para cada una ir sacando su valor, e imprimir ambos. Para obtener un listado de todas las claves, tienes el siguiente método en *Hashtable*:

Enumeration keys();

Devuelve una enumeración de las claves. Luego utilízala para recorrerlas, y con cada una sacar su valor e imprimirlo:

```
Enumeration en = ht.keys();
while (en.hasMoreElements()) {
  String clave = (String)(en.nextElement());
  String valor = (String)(ht.get(clave));
  ... // Imprimir clave y valor por pantalla
}
```

2. Comenta las conclusiones que obtienes tras haber hecho este ejercicio, y qué ventajas e inconvenientes encuentras a la hora de añadir elementos en la hash, y de recuperarlos.
3. Observa cómo se imprimen los valores de la hash al sacarlos por pantalla. ¿Conservan el orden en que se introdujeron? ¿A qué crees que puede deberse?
4. Si en lugar de trabajar con listas o tablas hash de 10 elementos fuesen de 1.000.000 de elementos, ¿quién se comportaría más eficientemente (*ArrayList* o *Hashtable*) y por qué? No hace falta que lo pruebes, haz una estimación basándote en lo que has visto hasta ahora.

EJ7.5-COL-FH

Ordenar tus propios datos

```
1   import java.util.*;
2
3   public class Persona {
4           String nombre;
5           String apellido1;
6           String apellido2;
7           String direccion;
8           String telefono;
9
10          public Persona (String nombre, String apellido1, String apellido2, String direccion, String telefono) {
11                  this.nombre = nombre;
12                  this.apellido1 = apellido1;
13                  this.apellido2 = apellido2;
14                  this.direccion = direccion;
15                  this.telefono = telefono;
16          }
17
18          public String getNombre() {
19                  return nombre;
20          }
21
22          public String getApellido1() {
23                  return apellido1;
24          }
25
26          public String getApellido2() {
27                  return apellido2;
28          }
29
30          public String getDireccion() {
31                  return direccion;
32          }
33
34          public String getTelefono() {
35                  return telefono;
36          }
37
38          public void setNombre(String nombre) {
39                  this.nombre = nombre;
40          }
41
42          public void setApellido1(String apellido1) {
43                  this.apellido1 = apellido1;
44          }
45
46          public void setApellido2(String apellido2) {
47                  this.apellido2 = apellido2;
48          }
49
50          public void setDireccion(String direccion) {
51                  this.direccion = direccion;
52          }
```

```
53
54          public void setTelefono(String telefono) {
55                  this.telefono = telefono;
56          }
57
58          public String toString() {
59                  return apellido1 + " " + apellido2 + ", " + nombre + ", " + direccion + ", " + telefono;
60          }
61
62          public static void main(String[] args) {
63                  ArrayList al = new ArrayList();
64
65                  al.add(new Persona("Marta", "García", "Hernández", "C/Aloma - 22", "634253456"));
66                  al.add(new Persona("Eva", "Simón", "Mas", "Camino del Prado - 30", "966124627"));
67                  al.add(new Persona("Rafael", "García", "Hernández", "C/Aloma - 1", "601123546"));
68                  al.add(new Persona("Manuel", "Bravo", "Murillo", "C/La Huerta - 22", "965123456"));
69                  al.add(new Persona("Carolina", "García", "Rodríguez", "Avda. Doctor Rico", "661228844"));
70
71                  Collections.sort(al);
72
73                  for (int i = 0; i < al.size(); i++) {
74                          System.out.println((Persona)(al.get(i)));
75                  }
76          }
77  }
```

La clase *Persona* de la plantilla almacena los datos generales de una persona, como son su nombre, primer apellido, segundo apellido, dirección y teléfono. Tiene un constructor que se encarga de asignar todos esos campos, y métodos *get()* y *set()* para obtener sus valores o cambiarlos, respectivamente.

Además, al final tiene un método *main* que crea varios objetos de tipo *Persona*, los coloca en un *ArrayList*, y luego intenta ordenarlos llamando al método *Collections.sort*. Sin embargo, de momento el método no funciona (probablemente salte una excepción, porque no sabe cómo comparar los elementos de la lista).

Haz las modificaciones necesarias en la clase para que el método ordene correctamente. Queremos que se siga el siguiente criterio de ordenación:

- Ordenar de menor a mayor, según el primer apellido.
- Si el primer apellido coincide, ordenar de menor a mayor según el segundo apellido.
- Si también coincide, ordenar de menor a mayor por el nombre. Si también coinciden, se considerará que los nombres son iguales en orden.

Comprueba, una vez lo tengas hecho, que la secuencia que saca el programa tras ordenar es la correcta:

Elemento 1: "Bravo Murillo, Manuel, C/La Huerta - 22, 965123456"
Elemento 2: "García Hernández, Marta, C/Aloma - 22, 634253456"
Elemento 3: "García Hernández, Rafael, C/Aloma - 1, 601123546"
Elemento 4: "García Rodríguez, Carolina, Avda. Doctor Rico - 25, 661228844"
Elemento 5: "Simón Mas, Eva, Camino del Prado - 30, 966124627"

Imagina que queremos cambiar el criterio de ordenación, y ahora queremos ordenar de mayor a menor por el nombre. ¿Qué cambios tendríamos que hacer? No los hagas, simplemente déjalos indicados en la respuesta a esta pregunta.

Tema 9: Excepciones

EJ9.1-EXC-OBE

Investigar la clase *Exception* del API de Java, qué métodos tiene y cuáles hereda.
¿Cuál es la salida de la siguiente aplicación?

```
1   import java.io.*;
2
3   public class Ej1 {
4
5       public static void main(String[] args) {
6           try {
7               throw new Exception("Mi nueva excepción");
8           }
9           catch(Exception e){
10              System.out.println("Dentro de catch");
11              System.out.println( "e.getMessage(): " + e.getMessage());
12              System.out.println( "e.toString(): " + e.toString());
13              System.out.println("e.printStackTrace():"); e.printStackTrace();
14          }
15      }
16  }
```

EJ9.2-EXC-OB

Se escribe la siguiente aplicación que permite leer una línea ingresada por el teclado la cual es impresa a continuación.

```
1   import java.io.*;
2   public class Ejercicio{
3
4     public static void main(String[] args) {
5       BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
6
7       System.out.print("Ingrese una línea:");
8
9       System.out.println(stdin.readLine());
10    }
11  }
12
```

El código no compila dando el siguiente mensaje:

> *Ej4.java:9: Exception java.io.IOException must be caught, or it must be declared in the throws clause of this method. System.out.println(stdin.readLine()); ^ 1 error*

System.out.println(stdin.readLine());

1 error

Arreglar la aplicación para que compile y pueda correr.

EJ9.3-EXC-OB

Implementar una aplicación para tratamiento de Fracciones que realice las siguientes operaciones:

a. Constructor
b. Acceso al numerador y al denominador
c. Operaciones de comparación de fracciones, tales como: igualdad, mayor, menor, mayor o igual y menor o igual.

Identifica los casos excepcionales en las operaciones y define las excepciones correspondientes

EJ9.4-EXC-FH

Implementar una aplicación para tratar la abstracción matemática Tiene que manejar conjuntos de números naturales del 1 al 100. Características:

- Descripción por enumeración, por ejemplo {1,4,6}
- No hay orden asociado.
- No hay elementos repetidos.
- No tiene sentido manipular los componentes individuales.

Operaciones:

- Crear el conjunto vacío.
- Insertar un elemento.
- Borrar un elemento.
- Igualdad entre conjuntos
- Comprobar si el conjunto está vacío, si un elemento pertenece a un conjunto y si dos conjuntos con iguales.
- Unión de conjuntos
- Intersección de conjuntos
- Resta de conjuntos
- Complementario de un conjunto
- Lectura y escritura de conjuntos en la entrada/salida estándar.
- Identifica los casos excepcionales en las operaciones y define las excepciones correspondientes

Se pide:

a. Dibuja un diagrama de clases que implementa la anterior interfaz.
b. Implementa la interfaz y la clase anteriores en Java.
c. Implementa un programa de prueba.

EJ9.5-EXC-OP

0.

```
1   void f1(int accion) throws EOFException {
2           try {
3                   if(accion == 1)
4                           throw new FileNotFoundException();
5                   else if(accion == 2)
6           throw new EOFException();
7           }
8           catch(FileNotFoundException e){
9                   System.out.println( “Error corregido” );
10          }
11          System.out.println( “Finalización normal de f1” );
12  }
13
14  void f2( int accion ){
15          try{
16                  f1( accion ):
17          }
18          catch( EOFException e ){
19                  System.out.println( “Error corregido” );
20          }
21          System.out.println( “Finalización normal de f2” );
22  }
```

¿Qué pasa si se llama a *f2()* pasándole los valores 1 ó 2 para el parámetro acción?
¿En qué método se producen las excepciones y en cuál se tratan?

1. Crear una clase con un método *main()* que lance un objeto de tipo Exception dentro de un bloque try. Dar al constructor de Exception un parámetro String. Capturar la excepción en una cláusula catch e imprimir el parámetro String. Añadir una cláusula finally e imprimir un mensaje para probar que uno paso por ahí.
2. Crear una clase de Excepción usando la palabra clave extends. Escribir un constructor para esta clase que tome un parámetro String y lo almacene dentro del objeto con una referencia String. Escribir un método que imprima el String almacenado. Crear una cláusula try-catch para probar la nueva excepción.
3. Escribir una clase con un método que lance una excepción del tipo creado en el Ejercicio 2. Intentar compilarla sin especificación de excepción para ver qué dice el compilador. Añadir la especificación apropiada. Probar la clase y su excepción dentro de una cláusula try-catch.
4. Definir una referencia a un objeto e inicializarla a null. Intentar llamar a un método mediante esta referencia. Ahora envolver el código en una cláusula try-catch para capturar la excepción.
5. Crear una clase con dos métodos, *f( )* y *g( )*. En *g( )*, lanzar una excepción de un nuevo tipo a definir. En *f( )*, invocar a *g( )*, capturar su excepción y, en la cláusula catch, lanzar una excepción distinta (de tipo diferente al definido). Probar el código en un método *main( )*.

6. Crear tres nuevos tipos de excepciones. Escribir una clase con un método que lance las tres. En el método *main( )*, invocar al método pero usar sólo una única cláusula catch para capturar los tres tipos de excepción.
7. Escribir código para generar y capturar una *ArrayIndexOutOfBoundsException*.
8. Crear tu propio comportamiento reiniciador utilizando un bucle while que se repita hasta que se deje de lanzar una excepción.
9. Crear una jerarquía de excepciones de tres niveles. Crear a continuación una clase base *A* con un método que lance una excepción a la base de la jerarquía. Heredar *B* de *A* y superponer el método de forma que lance una excepción en el nivel dos de la jerarquía. Repetir heredando la clase *C* de *B*. En el método *main( )*, crear un objeto de tipo *C* y hacer un conversión hacia arriba a *A*. Después, llamar al método.
10. Demostrar que un constructor de clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.
11. Mostrar que con *Finally.java* no falla lanzando una *RuntimeException* dentro de un bloque try.
12. Modificar el Ejercicio 6 añadiendo una cláusula finally. Mostrar que esta cláusula finally se ejecuta, incluso si se lanza una *NullPointerException*.

EJ9.6-EXC-OB

Escribe una clase llamada *DividePorCero* que pruebe a dividir un entero entre otro recogiendo la excepción de división por cero y mostrando el mensaje "*Error: división por cero*" si se produce.

Prácticas

P1-Bombillas

1. Crear una clase *Bombilla* que tenga como atributos un entero potencia, un String color y dos booleanos *encendida* y *fundida*.
2. Crear los métodos *get( )* y *set( )* [en clase *Bombilla*] de cada una de los atributos. Crear además un método *dameInfo( )* [en clase *Bombilla*] que nos muestre por consola la información de la bombilla, color, potencia, si está encendida o no.
3. Crear un método *encender( )* y otro *apagar( )* [en clase Bombilla] que cambie la variable encendida de *true* a *false* según sea el caso y saque un mensaje por consola que está encendida o apagada.
4. Los métodos *encender()* y *apagar()* tendrán que comprobar si la bombilla ya está encendida para apagarla o si ya está apagada para encenderla.
5. Crearemos la clase *AplicacionBombillas* donde pondremos el método *main*. Crearemos un objeto *Bombilla* y la encenderemos y la apagaremos, modificaremos sus atributos, etc.
6. Crearemos un método *fundir( )* [en clase *Bombilla*] que llamaremos desde el método *encender( )* para fundir las bombillas. Se generará un número aleatorio que nos dará la probabilidad de que se funda la bombilla. Tendremos que modificar los métodos *encender( )* y *apagar( )* ya que si la bombilla se ha fundido no se podrá encender ni apagar.
7. Crearemos otra clase con método *main* llamada *AplicaciónArrayBombillas* donde crearemos varios arrays de bombillas. Uno de forma estática de 5 bombillas, otro de forma dinámica de 3 bombillas y un tercero, estático, llamado *lucecillas* de 50 bombillas que rellenaremos con un bucle. Los otros dos arrays los rellenaremos manualmente.
8. Encenderemos y apagaremos cada una de las bombillas.
9. Crearemos otra clase con método *main* llamada *FundiendoBombillas*. Nos crearemos un array de 33 bombillas, estático, y lo rellenaremos con un bucle. Queremos que primero se enciendan todas las bombillas y luego se apaguen todas. El bucle tendrá que hacer esto hasta que estén todas fundidas.

P2-Arrays

1. Creamos una clase *Array1Dimension* con un método *main*.
    a. Creamos un array de 100 elementos de números generados aleatoriamente entre 1 y 100.
    b. Mostrar por consola todos los que son mayores de 33.
    c. Mostrar por consola la suma de todos los elementos del array.
2. Creamos una clase *Array1Dimension2* con un método *main*.
    a. Creamos un array de 30 elementos con números comprendidos entre 9 y -9 generados aleatoriamente.

    b. Mostrar por consola cuantos elementos de las posiciones pares son positivos, cuantos negativos y cuantos cero.
3. Creamos una clase *TresArrays* con un método *main*.
    a. Creamos dos *arrays* iguales al anterior (números entre -9 y 9) de 30 elementos.
    b. En un tercer array almacenamos la suma de los otros dos arrays elemento a elemento y lo mostramos por consola.
4. Creamos una clase *Array2Dimensiones* con un método *main*.
    a. Creamos un array bidimensional (5,10).
    b. Lo rellenamos con números generados aleatoriamente comprendidos entre 0 y 9.
    c. Mostramos por consola la suma de los elementos de cada fila del array.
    d. Mostramos por consola el valor mínimo de cada fila de la tabla.
5. Creamos una clase *ArrayAlumnos* con un método *main.* Crearemos un array con 20 alumnos. La primera columna de cada alumno será un número identificador del mismo que estaré comprendido entre 1000 y 1019. Tendrá tres columnas mas para cada una de las notas de las tres evaluaciones. Dichas notas las generaremos aleatoriamente.
    a. Visualizar por consola la media de cada uno de los alumnos.
    b. Visualizar por consola la nota media de toda la clase.
6. Crear una clase *NumerosPrimos* con un método *main*. Visualiza por consola todos los números primos entre 1 y 100.

P3- Estudiantes - CONSTRUCTORES

1. Crear una clase *Estudiantes* con los atributos:
    nombre
    apellido
    edad
2. Crear en dicha clase tres constructores, uno con todos los atributos, otro solo con nombre y apellido que llame al primero y fije la edad a 0 y un tercer constructor con solo el nombre que llame al segundo y fije el apellido a un valor “sin apellido”.
3. Crear un método *dameInfo()* que nos muestre información del *Estudiante*.
4. Crear una clase *AplicacionEstudiante* con un método *main* donde crearemos estudiantes con los atributos que nos parezcan.

P4-Colegio

1. Crear una clase Alumno que tendrá como atributos un String nombre y un entero edad. Crear su constructor con parámetros y otro sin parámetros.
2. Generar los métodos *get( )* y *set( )* y un método *dameInfo( )* que muestre por consola la información del alumno.
3. Crear una clase *Aula*, donde crearemos un array *estudiantes[ ]*. El constructor de la clase *Aula* recibirá ese array.

4. Dentro de esta clase crear un método *listar( )* que nos dé la información de todos los alumnos del aula.
5. Crear un método *añadirAlumno( )* con el que podremos añadir alumnos al array inicial (es decir, habrá que redimensionar el array).
6. Crear un método *borrarAlumno( )* con el que podremos borrar un alumno del aula (array) si existe y si no existe que nos avise de que el alumno no existe y no se puede borrar (es decir, habrá que redimensionar el array).
7. Crear una clase *Colegio* que contendrá un método *main*. En dicha clase crearemos alumnos y arrays de alumnos, *escolares[ ]*. Sacaremos sus atributos, se los cambiaremos. Crearemos un objeto *Aula* para almacenar el array de escolares, listaremos el array, añadiremos un alumno para luego borrarlo, y borraremos un alumno que estuviera inicialmente en el array.

P5-Hospital - HERENCIA

1. Crear una clase *Paciente* con los atributos: String *nombre*, int *vida*, boolean *curable*. Hacer métodos *get( )* y *set( )*, constructor con todos los atributos. En el método *setVida( )* hacer la comprobación para que no pueda tener una vida menor que 0.
2. Crear una clase *Medico* que tenga un método *consultar( )* que nos muestre el nombre y la vida del paciente al que le está pasando consulta. Cuando pasamos consulta comprobamos: si la vida es mayor que 2 ponemos atributo *curable* a *true*.
3. Crear clase *Cirujano* que herede de médico y que implemente el método *operar( )*. Dicho método recibe un paciente es curable o no. Si no es curable sacamos un mensaje diciendo que no le va a operar porque no se le puede curar. Si curable está a true generamos un número aleatorio. Si dicho número es menor que 3 el paciente se muere después de la operación. En caso contrario se recupera.
4. Crear una clase *Hospital* que tenga método *main* donde crearemos los objetos *paciente*, *médico* y *cirujano*. Haremos castings de *Medico* a *Cirujano* y viceversa para ver las diferencias.

P6-Libro de la selva - HERENCIA y POLIMORFISMO

1. Crear una clase *Animal* que tenga como atributos un String nombre y tres enteros *edad*, *peso* y *velocidad*. Crear su constructor y los métodos *get( )* y *set( ).* Implementar un método *comer( )* que nos muestre un mensaje por consola que diga “Como como un animal”.
2. Crear una clase *Humano*, crear su constructor, generar los métodos *get( )* y *set( )* y sobrescribir el método comer( ) para que diga algo propio de los humanos.
3. Crear una clase *Cebra*, crear su constructor, generar los métodos *get( )* y *set( )* y sobrescribir el método comer( ) para que diga algo propio de un rumiante.
4. Crear una clase *Mono*, crear su constructor, generar los métodos *get( )* y *set( )* y sobrescribir el método *comer( )* para que diga algo propio de un mono.
5. Crear una clase *León* que añada el atributo String *colorMelena*, crear su constructor, generar los métodos *get( )* y *set( )* y sobrescribir el método *comer( )* para que diga algo propio del León. Implementar el método *rugir( )* que se llamará

desde *comer( )* y un método *cazar( )* que reciba un *Animal* y saque un mensaje por consola de qué tipo de *Animal* ha cazado.

6. Crear una clase *LibroSelva* que contendrá el método *main* donde crearemos un array de *Animales* y llamaremos a los métodos *comer( ),* y *cazar( ).* El León podrá cazar a todos los animales excepto a otros leones.

P7- Empleados - HERENCIA

1. Crear una clase *Empleado* que tenga por atributos un String *nombre*, un String *apellidos* y un entero *edad*.
2. Generar su métodos *get( )* y *set( ),* un constructor con todos los parámetros, otro con sólo el nombre y los apellidos y un tercero sin parámetros.
3. Sobrescribir el método *toString( )* y escribir un método *iguales()* que compare empleados y vea si dos empleados son el mismo o no, es decir que compruebe que coinciden su nombre, sus apellidos y su edad.
4. Crear una clase *Programador* que extienda *Empleado* y tenga además el atributo entero *estres*. Hacer su constructor, método *get( )* y *set( )*, y sobrescribir los métodos *toString( )* e *iguales( ).*
5. Crear una clase *Secretaria* que extienda *Empleado* y tenga además el atributo String departamento. Hacer su constructor, método *get( )* y *set( )*, y sobrescribir los métodos *toString( )* e *iguales( ).*
6. Crear una clase *Becario* que extienda *Empleado* y tenga además el atributo entero numFotocopias. Hacer su constructor, método *get( )* y *set( )*, y sobrescribir los métodos *toString( )* e *iguales( ).*
7. Crear una clase *Jefe* que extienda *Empleado* y tenga además el atributo String departamento. Hacer su constructor, método *get( )* y *set( )*, y sobrescribir los métodos *toString( )* e *iguales( ).*Dentro de esta clase implementaremos un método *echarBronca( )* que echará broncas personalizadas, según el empleado sea un *Programador*, una *Secretaria* o un *Becario*. Tendremos que comprobar que el jefe tiene empleados a su cargo.
8. Cuando el *Jefe* echa una bronca a un *Programador* tendremos que subirle el estrés, cuando es a un *Becario* subirá el número de fotocopias que tiene que hacer, no podrá echar bronca a una *Secretaria* que no sea de su departamento y cuando se dirija a otro jefe será amigable.
9. Crearemos una clase *Oficina* que contendrá el método *main* donde crearemos un array de empleados de distintas clases. Mostraremos las características de cada uno y un crearemos un jefe que les echará la bronca.

P8-Cielo - INTERFACES

1. Crear una interface *IVolar* que declare el método *volar( )*.
2. Crear una clase *Superman* que tenga por atributo un entero fuerza e implemente la interface *IVolar*. Crear su constructor y generar los métodos *get( )* y *set( ).* Implementar el método *volar( )* que sacará un mensaje por consola diciendo que

está volando. Además cada vez que *Superman* vuele perderá fuerza y cuando la fuerza sea menor que 0 sacará un mensaje de que no puede volar.

3. Crear una clase *Avion* que tenga por atributo un entero combustible e implemente la interface *IVolar*. Crear su constructor y generar los métodos *get( )* y *set( )*. Implementar el método *volar( )* que sacará un mensaje por consola diciendo que está volando. Además cada vez que el *Avión* vuele gastará combustible y cuando el combustible sea menor que 15 sacará un mensaje de que no puede volar.
4. Crear una clase *Pajaro* que tenga por atributos un entero *envAlas* y otro cansancio e implemente la interface *IVolar*. Crear su constructor y generar los métodos *get( )* y *set( )*. Implementar el método *volar( )* que sacará un mensaje por consola diciendo que está volando. Además cada vez que el *Pájaro* vuele ganará cansancio y cuando su cansancio sea mayor que 20 sacará un mensaje de que no puede volar.
5. Crear una clase *Cielo* que contendrá el método *main* desde la cual crearemos a *Superman*, *pájaros* y *aviones* y los haremos *volar( )*.