

APRENDER PROGRAMACIÓN JAVA DESDE CERO.

[PROGRAMACIÓN ORIENTADA A OBJETOS]

Objetivos

Java es uno de los lenguajes de programación más utilizados en el mundo, enmarcado en el grupo de lenguajes orientados a objetos. Este curso permite aprender los fundamentos de la programación Java y de la programación orientada a objetos.

Destinatarios

Cualquier persona con interés en aprender fundamentos de programación Java con vistas al desarrollo de aplicaciones. Es recomendable, aunque no imprescindible, que el alumno tenga conocimientos básicos de algoritmia y de algún otro lenguaje de programación.

Contenidos

- ✓ INTRODUCCIÓN A JAVA. QUÉ ES JAVA. INSTALACIÓN Y PRIMEROS PASOS CON JAVA.
- ✓ OBJETOS, CLASES Y CONSTRUCTORES EN JAVA. INSTANCIAS. TIPOS DE DATOS.
- ✓ SINTAXIS BÁSICA Y CONDICIONALES EN JAVA. OPERADORES.
- ✓ EL API DE JAVA. BIBLIOTECAS DE CLASES. ¿QUÉ ES Y PARA QUÉ SIRVE EL API DE JAVA?
- ✓ CREAR UN PROGRAMA CON JAVA. ABSTRACCIÓN. MODULARIZACIÓN. MÉTODOS. MAIN.
- ✓ ESTRUCTURAS DE REPETICIÓN O BUCLES, COLECCIONES DE OBJETOS Y RECORRIDOS.
- ✓ HERENCIA EN JAVA. ¿QUÉ ES LA HERENCIA EN PROGRAMACIÓN ORIENTADA A OBJETOS?
- ✓ PROGRESAR COMO PROGRAMADORES JAVA: SWING, GESTIÓN DE ERRORES Y OTROS

Duración

150 horas de dedicación efectiva, incluyendo lecturas, estudio y ejercicios.

Dirección, modalidades y certificados

El curso está dirigido por Mario Rodríguez Rancel, Jefe de Proyectos de aprenderaprogramar.com. Se oferta bajo las modalidades web (gratuito), con tickets de soporte y tutorizado on-line (material + soporte). A los alumnos que sigan el curso tutorizado on-line y cumplan el programa de trabajo se les expedirá certificado acreditativo de la realización del curso.

INDICE DEL CURSO

1. CUESTIONES BÁSICAS SOBRE JAVA

- 1.1. ¿Qué es Java?
- 1.2. ¿Es Java un lenguaje ideal para aprender a programar?
- 1.3. ¿Es Java software libre?
- 1.4. ¿Cuáles son las versiones y distribuciones Java y cuál usar?
- 1.5. ¿Qué sistema operativo debo usar para programar Java?
- 1.6. ¿Qué son las actualizaciones de Java?

2. INSTALAR JAVA EN NUESTRO ORDENADOR Y DAR LOS PRIMEROS PASOS

- 2.1. Descargar (download) Java: obtener el instalador
- 2.2. Instalación de Java en Windows (en nuestro pc)
- 2.3. Configuración de Java en Windows: JAVA_HOME, PATH
- 2.4. Máquina virtual JVM, compilador e intérprete Java. Bytecode.
- 2.5. NetBeans, Eclipse, JCreator, JBuilder... ¿Cuál es el mejor entorno de desarrollo (IDE) Java?
- 2.6. Descargar (download) BlueJ. Instalación en Windows.
- 2.7. ¿Qué es un proyecto Java? Creación del primer proyecto.
- 2.8. La ventana del editor de BlueJ
- 2.9. Escribir código (una clase) en Java con un IDE. Primer programa.
- 2.10. Comentarios en lenguaje Java y bloques en Java

3. CONCEPTOS BÁSICOS DE PROGRAMACIÓN ORIENTADA A OBJETOS. CLASES Y OBJETOS.

- 3.1. Visualizar clases y objetos con BlueJ e invocar métodos
- 3.2. Tipos de datos (variables) en Java
- 3.3. Ejemplos de uso de tipos de datos (variables) en Java. Declaración y asignación de contenido.
- 3.4. ¿Qué es una clase? Atributos, constructor y métodos
- 3.5. Métodos tipo procedimiento (...void) y tipo función (...return).
- 3.6. Métodos con y sin parámetros
- 3.7. Métodos consultores o de acceso (getters) y métodos modificadores (setters)
- 3.8. Estado de un objeto
- 3.9. Parámetros formales y parámetros actuales
- 3.10. Comprender la filosofía de métodos y clases en Java
- 3.11. Signatura de un método. Interfaz o interface.
- 3.12. Guardar los proyectos Java. Copias de seguridad.

4. ESCRITURA BÁSICA DE CÓDIGO Y ESTRUCTURAS DE DECISIÓN

- 4.1 Imprimir por pantalla con System.out. Concatenar cadenas. Notación de punto.
- 4.2 Operadores aritméticos en Java.
- 4.3 Operadores lógicos principales en Java.
- 4.4 Sentencia de asignación. Asignación compuesta.
- 4.5 Condicional if else / if else if en Java
- 4.6 Condicional de selección con switch en Java

5. ESTRUCTURAR EL CÓDIGO EN JAVA

- 5.1 Variables locales a un método o constructor. Sobre carga de nombres.
- 5.2 Cómo crear constructores en Java. Ejemplos.
- 5.3 Clases con dos o más constructores. Sobre carga de constructores o métodos.
- 5.4 Clases que utilizan objetos preexistentes: relación de uso entre clases. Diagrama de clases.
- 5.5 Paso de objetos como parámetros a un método o constructor
- 5.6 La sentencia new como invocación de un constructor.

6. LIBRERÍAS DEL API DE JAVA. BIBLIOTECAS DE CLASES.

- 6.1 Otra definición de clase: un paquete de código. Objetos del mundo real y objetos abstractos.
- 6.2 ¿Qué es y para qué sirve el API de Java?
- 6.3 Organización y forma de nombrar las librerías en el API de Java
- 6.4 Importar y usar clases del API de Java. Ejemplo: clase Math
- 6.5 Interface de clase. Ejemplo: clase String
- 6.6 Explorar un método. El método substring de la clase String.
- 6.7 Usar métodos para evitar errores. Ejemplo método length de la clase String

7. CONSTRUIR UN PROGRAMA CON JAVA

- 7.1 Concepción de programas mediante abstracción y modularización
- 7.2 Un ejemplo de código Java muy elemental. Crear clases con campos, constructor y métodos.
- 7.3 Definición de método interno y método externo en Java
- 7.4 La palabra clave this. Contenido null por defecto de un objeto.
- 7.5 Clase con el método main: clase principal, iniciadora o “programa” principal.
- 7.6 Sintaxis y código ejemplo de uso del método main.
- 7.7 Pedir datos por consola (teclado) en java. Escape con barra invertida backslash. Print y salto de línea con \n.

8. BUCLES, COLECCIONES DE OBJETOS Y RECORRIDOS DE COLECCIONES

- 8.1 Concepto general de bucle
- 8.2 Bucle con instrucción for. Operador ++ y --. Sentencia break
- 8.3 Bucle con instrucción while. Ejemplo de uso de break.
- 8.4 Bucle con instrucción do ... while.
- 8.5 El debugger de BlueJ. Detener un programa en ejecución.

- 8.6 Pensar en objetos. Pensar una entrada de teclado como objeto.
- 8.7 El método equals. Diferencia entre igualdad e identidad entre objetos.
- 8.8 Asignación de igualdad con tipos primitivos y con objetos.
- 8.9 Repaso y ejemplos sobre igualdad, identidad y método equals.
- 8.10 Colecciones de objetos de tamaño flexible. Contenedores de objetos.
- 8.11 La clase ArrayList del API de Java.
- 8.12 Concepto de clase genérica (clase parametrizada)
- 8.13 El for extendido o bucles for each en Java.
- 8.14 Recorrer colecciones con objetos Iterator. Evitar errores tipo java.util.ConcurrentModificationException.
- 8.15 Resumen de tipos de bucles (ciclos) en Java.
- 8.16 Objetos con referencia null y excepciones tipo java.lang.NullPointerException.
- 8.17 Autoboxing y unboxing. Conversión automática de tipos envoltorio a primitivos y viceversa.
- 8.18 Objetos anónimos.
- 8.19 Colecciones de tamaño fijo: arrays, arreglos o formaciones.
- 8.20 Campo length para saber el número de elementos de un array
- 8.21 Uso de ciclos for each con arrays
- 8.22 Resumen de colecciones

9. MÁS CONCEPTOS Y CUESTIONES BÁSICAS DE JAVA

- 9.1 Conversión de tipos en Java
- 9.2 Método valueof para conversión de tipos
- 9.3 Ejemplo de conversión de tipos. Tipo obtenido frente a tipo requerido. Métodos get y remove de ArrayList
- 9.4 Generar números aleatorios en Java. Clase Random.
- 9.5 Variables de clase o estáticas y constantes. Palabras clave static y final.
- 9.6 Organizar un proyecto Java en paquetes (packages). Cláusulas package e import
- 9.7 Formas de nombrar packages, jerarquización y visibilidad de clases. Los packages en BlueJ.
- 9.8 Copiar arrays y comparar arrays. Identidad e igualdad entre arrays.
- 9.9 La clase Arrays del API de Java. Métodos equals y copyof.
- 9.10 Rellenar un array con un valor u objeto. Método fill de la clase Arrays.
- 9.11 Interfaz o interface en Java. Ampliación del concepto.
- 9.12 Polimorfismo en Java. Primera aproximación.
- 9.13 Transformar un array en una lista con el método aslist de la clase Arrays. Constructores que usan colecciones.
- 9.14 Documentar un proyecto con javadoc. Comentarios, símbolos, tags.
- 9.15 Tipos enumerados (enum) en Java.
- 9.16 Método values. Enumerados clases con campos y constructores.
- 9.17 Métodos de clase o static frente a métodos de instancia. Comprender el método main.

10. HERENCIA EN JAVA. ¿QUÉ ES? ¿PARA QUÉ SIRVE?

- 10.1 ¿Qué es la herencia en programación orientada a objetos?
- 10.2 Jerarquías de herencia. Organización y acceso entre clases.

- 10.3 Ejemplo de herencia y uso de palabras clave extends y super. Constructores con herencia.
- 10.4 Ejemplo de herencia descendente o herencia simple.
- 10.5 Jerarquía de tipos. Subtipos. Polimorfismo y variables polimórficas.
- 10.6 Conversión de tipos (enmascaramiento). Hacer casting y ClassCastException.
- 10.7 Determinación del tipo de variables con instanceof.
- 10.8 Tipo estático y tipo dinámico de variables. Sobreescritura (redefinición) de métodos. Métodos polimórficos.
- 10.9 Ejercicio ejemplo de código con herencia, polimorfismo de variables y métodos, y sobreescritura de métodos.
- 10.10 Uso de la palabra clave super para llamar a métodos.
- 10.11 Modificadores de acceso public, private y protected.
- 10.12 Sobreescribir métodos de la clase Object: método toString.
- 10.13 Sobreescribir métodos de la clase Object: método equals.
- 10.14 Clases y métodos abstractos.
- 10.15 Clases abstractas en el API de Java
- 10.16 Herencia múltiple. Interfaces en Java.
- 10.17 Para qué sirven las interfaces en Java.
- 10.18 Ejemplo sencillo de interface en Java.
- 10.19 Implementar una interface del API de Java.
- 10.20 Resumen de herencia en Java.

11. IR MÁS ALLÁ EN JAVA: SWING, GESTIÓN DE ERRORES Y MÁS.

- 11.1 ¿Qué hemos aprendido y qué no hemos aprendido con este curso?

ORIENTACIÓN SOBRE EL CURSO PASO A PASO “APRENDER A PROGRAMAR EN JAVA DESDE CERO”

Java es un lenguaje muy potente de amplio uso a nivel profesional y empresarial. Este curso, que estamos comenzando, va dirigido a aquellas personas que quieran adquirir unos fundamentos serios de Java con vistas a poder desarrollar en el futuro aplicaciones atractivas y con cierta complejidad. No vamos a desarrollar un manual de referencia Java, sino un curso básico paso a paso. No vamos a contemplar todos los aspectos del lenguaje Java, sino aquellos que consideramos básicos desde el punto de vista didáctico, con vistas a que posteriormente la persona que lo deseé amplíe sus conocimientos. Nuestro objetivo es ser **claros, sencillos y breves**, y para eso tenemos que centrarnos en determinadas cuestiones de Java y dejar de lado otras.

Como conocimientos previos para iniciar este curso recomendamos (seguir la recomendación o no queda a criterio del alumno y/o profesor que vayan a seguir el curso) los siguientes: Algoritmia básica y fundamentos de programación, Lenguaje de programación Visual Basic ó C/C++ y Ofimática básica. Todos estos conocimientos previos están disponibles en aprenderaprogramar.com, en concreto a través de los siguientes cursos:

- Fundamentos de la Programación nivel I. Bases y Pseudocódigo.
- Fundamentos de la Programación nivel II. Programación modular.
- Programación en Visual Basic nivel I.

Los conocimientos previos son, como hemos dicho, deseables pero no imprescindibles.

Aprender programación Java requiere tiempo y esfuerzo. Para hacer ese recorrido más llevadero, te recomendamos que utilices los foros de aprenderaprogramar.com, herramienta a disposición de todos los usuarios de la web (<http://www.aprenderaprogramar.com/foros/>), y que te servirá para consultar dudas y recabar orientación sobre cómo enfrentarte a los contenidos. Entre los miembros del portal web y otros usuarios, trataremos de ayudarte para que el estudio te sea más llevadero y seas capaz de adquirir los conocimientos necesarios y avanzar como programador.

El tiempo necesario (orientativamente) para completar el curso incluyendo prácticas con ordenador, suponiendo que se cuenta con los conocimientos previos necesarios, se estima en 150 horas de dedicación efectiva o aproximadamente dos meses y medio con una dedicación de 3 horas diarias de lunes a viernes. Aprender programación requiere dedicación y esfuerzo.

El curso ha sido generado paso a paso usando Windows como sistema operativo y por ello contiene algunas indicaciones específicas para usuarios de Windows, pero también puede ser utilizado en otros entornos (Linux, Macintosh, etc.).

Una vez completado el curso, puedes profundizar en Java a través de contenidos complementarios que se ofrecen en aprenderaprogramar.com.

Estamos seguros de que con tu esfuerzo y la ayuda que te podamos brindar este curso te resultará de gran utilidad.

Próxima entrega: CU00603B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ ES JAVA? ¿ES NECESARIO SABER C ó C++ PARA PROGRAMAR EN JAVA?

Java es un lenguaje de programación orientado a objetos que se popularizó a partir del lanzamiento de su primera versión comercial de amplia difusión, la JDK 1.0 en 1996. Actualmente es uno de los lenguajes más usados para la programación en todo el mundo.

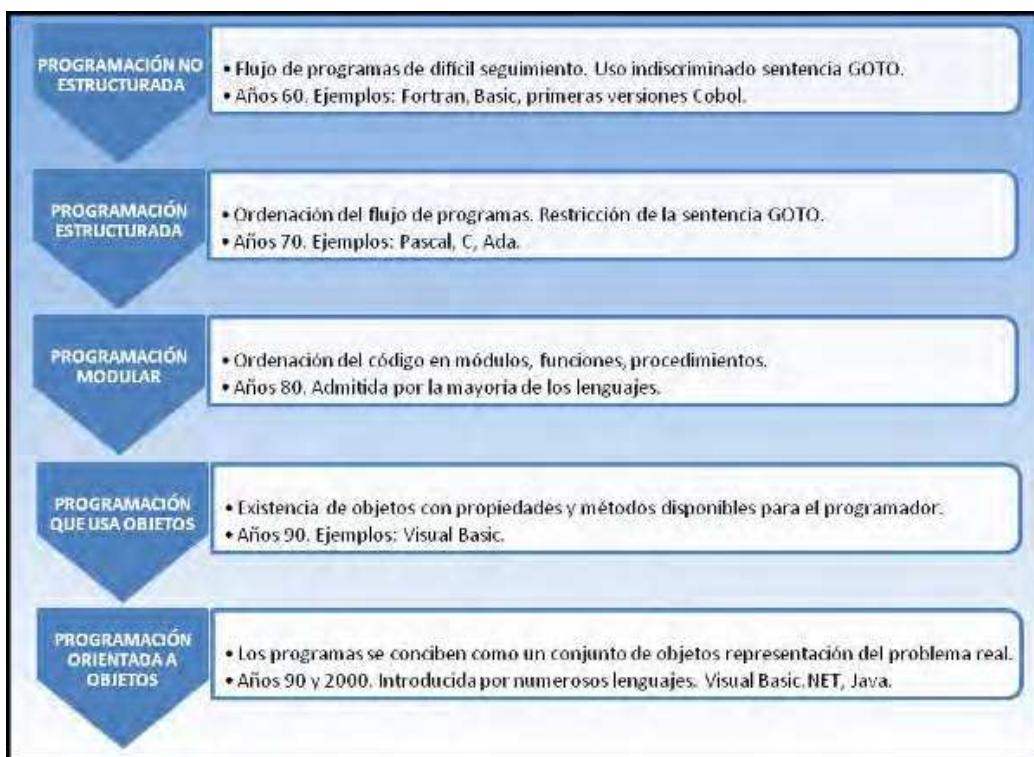


Los antecedentes de Java habría que buscarlos en los lenguajes de programación C y C++. El lenguaje C fue desarrollado en la década de los 70 y constituye un lenguaje muy robusto usado como núcleo del sistema operativo Unix. C no admite la orientación a objetos y está considerado un lenguaje “poco amigable” desde el punto de vista de que su sintaxis, elementos de programación que emplea (manejo directo de memoria) y otras cuestiones hacen que sea un lenguaje difícil de aprender. C++ fue una evolución de C desarrollada en los años 80. Introdujo el diseño orientado a objetos, pero manteniendo la compatibilidad con C. Esto suponía que C++ permitiera tanto el uso de la programación estructurada “tradicional” como la programación orientada a objetos. Además C++ mantuvo ciertas características de C como el manejo directo de la memoria, el uso de variables globales, sentencia goto, etc. que hicieron que la evolución fuera “parcial”.

Como paso final en esta evolución tenemos Java, un lenguaje que evoluciona a partir de C y C++, pero que elimina diversos aspectos de estos lenguajes y se constituye en un lenguaje definitivamente orientado a objetos. El romper con distintos aspectos de C++ cuyo manejo inadecuado por parte de muchos programadores daba lugar a problemas en las aplicaciones ha sido un factor decisivo para convertir a Java en un lenguaje popular y de amplio uso.

Nosotros vamos a quedarnos con el lado práctico de lo que hemos comentado respondiendo esta pregunta: **¿Es necesario saber C ó C++ para programar en Java?** No, no es necesario. Aunque puede suponer una ventaja para aquellas personas que tengan conocimientos previos en estos lenguajes, no recomendamos de forma explícita su estudio en profundidad como paso previo al aprendizaje de Java.

De modo orientativo, veamos un esquema sobre la evolución de los lenguajes.



Este esquema es meramente orientativo: es imposible reflejar la diversidad de lenguajes y su evolución en un gráfico tan simplificado. También los datos relativos a fechas son orientativos.

No hay que suponer que lo único válido sea la programación orientada a objetos por ser lo más moderno. Al contrario, muchísima programación de la que se hace hoy en día se basa en lenguajes o código no orientado a objetos. Además, la misma programación orientada a objetos se basa en conceptos muy antiguos de programación.

Tener en cuenta que algunos lenguajes que nacieron en los años 60 han perdido vigencia y ya no se usan, mientras que otros se han ido modernizando y continúan usándose más o menos ampliamente, como es el caso de Cobol. Lo que consideramos interesante con este esquema es que se vea que Java es una evolución que por un lado incorpora cosas que se venían usando desde hace mucho tiempo en programación, y por otro introduce ciertas novedades que lo convierten en un lenguaje moderno.

Java es un lenguaje útil para casi todo tipo de problemas. Podemos citar como funcionalidades de Java varias:

1. Aplicaciones “cliente”: son las que se ejecutan en un solo ordenador (por ejemplo el portátil de tu casa) sin necesidad de conectarse a otra máquina. Pueden servirte por ejemplo para realizar cálculos o gestionar datos.

2. Aplicaciones “cliente/servidor”: son programas que necesitan conectarse a otra máquina (por ejemplo un servidor de datos) para pedirle algún servicio de forma más o menos continua, como podría ser el uso de una base de datos. Pueden servir por ejemplo para el teletrabajo: trabajar desde casa pero conectados a un ordenador de una empresa.

3. Podemos hablar también de “**aplicaciones web**”, que son programas Java que se ejecutan en un servidor de páginas web. Estas aplicaciones reciben “solicitudes” desde un ordenador y envían al navegador (Internet Explorer, Firefox, Safari, etc.) que actúa como su cliente páginas de respuesta en HTML.

Éstos son sólo algunos ejemplos de todo el potencial que hay detrás de Java como lenguaje para aprender y obtener muchos beneficios con su uso. Obviamente por determinados términos empleados (cliente, cliente/servidor, base de datos, HTML...), te darás cuenta de que el lenguaje Java tiene mucha potencialidad, pero también de que su conocimiento a fondo requeriría mucho tiempo. Nosotros en este curso vamos a estudiar únicamente los aspectos más básicos de Java.

No te preocupes si no has entendido todo lo expuesto hasta ahora. Nuestra metodología se va a basar en ir trabajando poco a poco con Java e ir aprendiendo gradualmente mediante el trabajo práctico. Por otro lado, cuando se habla de Java es habitual oír hablar de definiciones teóricas sobre qué es la programación orientada a objetos, sus características (herencia, abstracción, polimorfismo, encapsulamiento), los objetos, la máquina virtual Java, y siglas como JVM, JRE, JDK, etc. Nosotros no vamos a hacer un recorrido teórico por estos conceptos: buscamos un recorrido práctico. Para ello vete leyendo el texto y probando en tu ordenador los ejemplos, ejercicios o programas que iremos mostrando o proponiendo.

Próxima entrega: CU00604B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿ES JAVA EL MEJOR LENGUAJE PARA APRENDER PROGRAMACIÓN SIN CONOCIMIENTOS PREVIOS?

Qué lenguaje es el más adecuado para aprender a programar es un tema de discusión entre programadores, profesores de universidad, profesionales, etc. La cuestión es que no hay un criterio unánime respecto a qué lenguaje es el ideal para aprender como primer lenguaje, posiblemente porque no existe ninguno ideal.



Haciendo una analogía, podría ser como tratar de responder a la pregunta: ¿Qué vehículo es el ideal para aprender a conducir? Y dado la diversidad de marcas (p.ej. Renault, Ford, Toyota, etc.), tipos de motor (diesel, gasolina), gestión de marchas (manual, automática), tecnología (híbrida, microhíbrida, convencional), y aún de tipos de vehículo (camiones, furgonetas, todoterrenos, turismos, etc.) sería muy difícil que hubiera un acuerdo unánime respecto a qué vehículo en concreto es el más adecuado porque existen cientos de posibilidades. En programación nos encontramos con que ocurre algo parecido: lenguajes fuertemente tipados o no tipados, orientados a objetos o no, diferentes versiones, sistemas operativos, filosofías de programación, de alto nivel o bajo nivel, y un sinfín de variantes que hacen que una persona que se plantee aprender a programar no lo tenga sencillo para elegir.

Nosotros vamos a expresar nuestra opinión: Java no es el lenguaje más adecuado para aprender como primer lenguaje de programación. Los argumentos para ello son los siguientes:

- a) Java se basa en una filosofía de programación (la orientación a objetos) que es una evolución de otras formas de entender la programación como la programación modular.
- b) Java es un lenguaje que conceptualmente a nivel de organización del código y recursos para el programador (API) puede resultar difícil para personas que se enfrentan a la programación por primera vez.
- c) Java tiene sus orígenes en otros lenguajes de programación como C y C++.
- d) Java lo consideramos un lenguaje que no es fácil de aprender si lo comparamos con otros lenguajes, de acuerdo con nuestra experiencia. Para una persona que empieza, puede ser más difícil el desarrollo de una primera aplicación con aspecto profesional en Java que en otros lenguajes.

Siguiendo con nuestra analogía con los tipos de vehículo, para nosotros Java vendría siendo un todoterreno de tecnología avanzada con diferentes opciones de tracción (a dos ruedas, a cuatro ruedas). Nosotros no seríamos partidarios de enseñar a conducir a una persona con este tipo de vehículo: por el contrario, nos decantaríamos por un vehículo más sencillo y manual como primera opción. La pregunta podría ser ahora: **¿es necesario empezar a trabajar con los lenguajes primitivos para ir avanzando a partir de ellos?** En nuestra opinión no, tampoco es necesario retrotraerse a los años 60 y recorrer los distintos lenguajes y filosofías hasta llegar a Java u otro lenguaje de última generación.

Entonces, ¿qué lenguaje elegir? He aquí el dilema. Los lenguajes C/C++ se adaptarían en nuestro símil a la consideración de “manual”, pero no a la de sencillos. Otros lenguajes como Visual Basic se adaptarían

a la consideración de sencillos, pero serían “semi-automáticos”. Y es aquí a donde queremos llegar: nosotros recomendamos Visual Basic como primer lenguaje de programación porque reúne características como sencillez, modernidad, facilidad de aprendizaje y para el desarrollo temprano de aplicaciones de aspecto profesional. Nos permite un recorrido por las bases de la programación al tiempo que introduce objetos con atributos y métodos de forma “moderada”.

Los motivos expuestos nos llevan a que no consideremos Java un lenguaje ideal para comenzar con la programación a personas que no tienen absolutamente ningún conocimiento previo, aunque con esto no queremos decir que Java no pueda ser usado como primer lenguaje en algunas circunstancias. De hecho, muchas universidades utilizan Java como primer lenguaje de programación en los estudios de Ingeniería Informática. Otras universidades incluyen asignaturas introductorias previas a Java basadas en C/C++.

Y a todas estas, ¿cuál es la conclusión? Que recomendamos que se tengan conocimientos de algún otro lenguaje de programación (el que sea), o al menos de pseudocódigo, antes de enfrentarse a Java. Seguir esta recomendación o no queda a la elección de cada cual, en base al tiempo disponible, objetivos personales, profesionales, académicos, etc. En este curso **vamos a explicar Java desde cero**, lo que significa que podrá seguirlo cualquier persona independientemente de sus conocimientos previos. Si después de lo expuesto aún te quedan dudas, puedes escribir tu consulta en los foros de aprenderaprogramar.com donde el staff de la página y otros usuarios podrán darte una orientación personalizada para tu caso en concreto.

Próxima entrega: CU00605B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿ES JAVA SOFTWARE LIBRE?

Considerar Java software libre no es del todo correcto. Pero considerarlo software propietario quizás tampoco lo es. Podríamos considerar entonces Java como una tecnología semi-liberada. No vamos a entrar en la discusión de si la política que sigue la empresa desarrolladora con Java es adecuada o no, ni en qué categoría de software libre o no libre podría clasificarse.



Nos vamos a quedar con el lado práctico. En este sentido, podemos acceder a todo lo necesario para programar en Java de forma gratuita: compilador, máquina virtual, biblioteca de clases, etc. están disponibles gratuitamente. Podemos desarrollar nuestros programas en Java, e incluso disponer de magníficos entornos de desarrollo de tipo profesional de forma gratuita. Solo a los programadores muy avanzados y amantes del software libre puro les puede resultar "desagradable" no poder acceder al código fuente del API de Java y crear sus propias distribuciones de Java. Las personas que quieren aprender Java o desarrollar programas con Java en general tendrán más que suficiente con todas las posibilidades gratuitas que existen en torno a esta tecnología. Así pues, no tengas ninguna preocupación pues se puede aprender Java con muchísimas herramientas y utilidades disponibles de forma gratuita.

¿QUÉ SISTEMA OPERATIVO ES MEJOR PARA PROGRAMAR CON JAVA?

Una de las ventajas de Java es que es multiplataforma: puedes usar cualquier sistema operativo como Windows, Linux, Macintosh, etc. Nosotrosaremos referencia con mayor frecuencia a Windows por ser el sistema más usado, pero puedes usar indistintamente uno u otro sin ningún problema.

Próxima entrega: CU00606B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

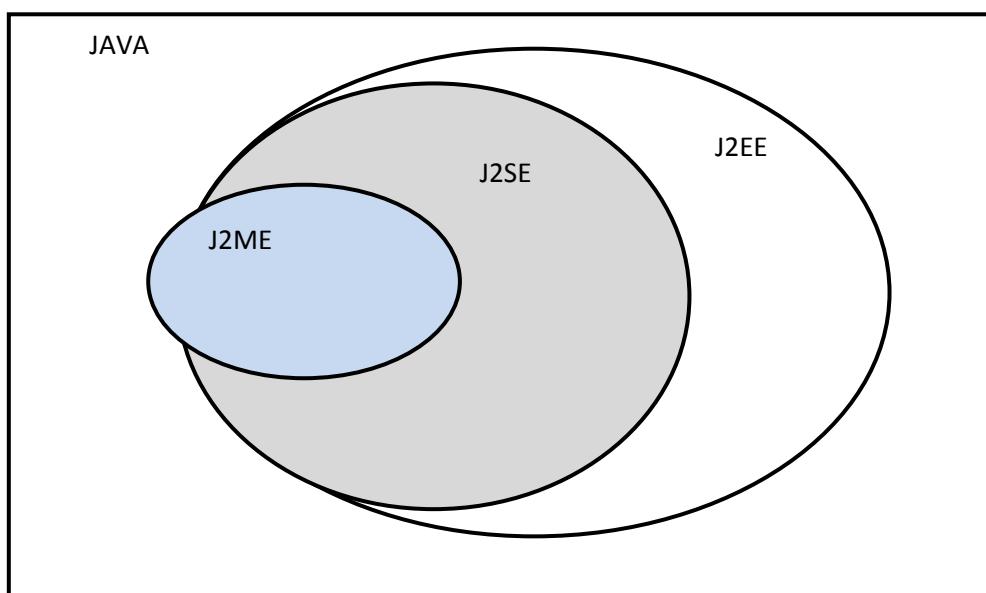
VERSIONES Y DISTRIBUCIONES DE JAVA

Java, como la mayoría de los lenguajes, ha sufrido diversos cambios a lo largo de su historia. Además, en cada momento han coexistido distintas versiones o distribuciones de Java con distintos fines. Actualmente puede considerarse que el Java vigente se denomina Java 2 y existen 3 distribuciones principales de Java 2, con ciertos aspectos comunes y ciertos aspectos divergentes.



Estas tres distribuciones son:

- a) **J2SE o simplemente Java SE:** Java 2 Standard Edition o Java Standard Edition. Orientado al desarrollo de aplicaciones cliente / servidor. No incluye soporte a tecnologías para internet. Es la base para las otras distribuciones Java y es la plataforma que utilizaremos nosotros en este curso por ser la más utilizada.
- b) **J2EE:** Java 2 Enterprise Edition. Orientado a empresas y a la integración entre sistemas. Incluye soporte a tecnologías para internet. Su base es J2SE.
- c) **J2ME:** Java 2 Micro Edition. Orientado a pequeños dispositivos móviles (teléfonos, tabletas, etc.).



En esta imagen vemos, de forma orientativa, como J2EE “expande” a J2SE, mientras que J2ME “recorta” a J2SE al tiempo que tiene una fracción de contenido diferenciada exclusiva de J2ME. En realidad hablar de expansiones y recortes no es correcto, porque cada distribución es en sí misma distinta puesto que están concebidas con distintas finalidades. Por tanto no puede decirse que sean expansiones o recortes, pero de forma coloquial muchas veces se interpreta así.

Java hoy en día es más que un lenguaje de programación, como veremos más adelante. El lenguaje Java estándar ha experimentado numerosos cambios desde la versión primigenia, JDK 1.0, así como un enorme incremento en el número de recursos disponibles para los programadores Java. Podemos citar en la evolución del Java estándar:

- **JDK 1.0** (1996): primer lanzamiento del lenguaje Java.
- **JDK 1.1** (1997): mejora de la versión anterior.
- **J2SE 1.2** (1998): ésta y las siguientes versiones fueron recogidas bajo la denominación Java 2 y el nombre "J2SE" (Java 2 Platform, Standard Edition), reemplazó a JDK para distinguir la plataforma base de J2EE (Java 2 Platform, Enterprise Edition) y J2ME (Java 2 Platform, Micro Edition). Incluyó distintas mejoras.
- **J2SE 1.3** (2000): mejora de la versión anterior.
- **J2SE 1.4** (2002): mejora de la versión anterior.
- **J2SE 5.0** (2004): originalmente numerada 1.5, esta notación aún es usada en ocasiones. Mejora de la versión anterior.
- **Java SE 6** (2006): en esta versión, Sun cambió el nombre "J2SE" por Java SE y eliminó el ".0" del número de versión. Mejora de la versión anterior.
- **Java SE 7** (2011): nueva versión que mejora la anterior. Incluyó mayor soporte para XML.
- **Java SE 8** (2014): nueva versión que mejora la anterior. Incluye la posibilidad de embeber JavaScript con Java y mejoras en la gestión de fechas y tiempo.
- **Java SE 9**: nueva versión que mejora la anterior (en difusión).
- **Java SE 10**: nueva versión que mejora la anterior (todavía sin uso comercial).

En Java todas las versiones siguen los mismos estándares de datos, esto permite que un programa que hayamos hecho con una versión antigua, pueda ser ejecutado con una versión más nueva sin necesidad de ningún cambio.

Además de los cambios en el lenguaje en sí, con el paso de los años los recursos disponibles para los programadores Java que ofrece la empresa que desarrolla el lenguaje (antiguamente Sun Microsystems, actualmente Oracle) han crecido enormemente. La denominada "biblioteca de clases de Java" (Java class library) ha pasado de ofrecer unos pocos cientos de clases en JDK 1.0 hasta cerca de 6000 en Java SE 8. Se han introducido recursos completamente nuevos, como Swing y Java2D, mientras que muchos de los métodos y clases originales de JDK 1.0 han dejado de utilizarse.

Cuando trabajamos con Java será frecuente que busquemos información "oficial" en internet. Cuando decimos oficial nos referimos a la que ofrece la propia empresa desarrolladora de Java. Cuando buscamos información sobre Java hay que tener cuidado respecto a a qué versión hace alusión la información. Por ejemplo, prueba a buscar "ArrayList java" o "ArrayList api java" en google, yahoo, bing o cualquier otro buscador. Un resultado posible es el siguiente (fíjate que en un caso es Java 1.4 y en otro Java SE 7):

- [ArrayList \(Java 2 Platform SE v1.4.2\)](#)

*java.util. Class ArrayList. java.lang.Object extended by java.util.AbstractCollection extended by java.util.AbstractList extended by ...
download.oracle.com/javase/.../java/.../ArrayList.html - [En caché](#) - [Similares](#)*

- [ArrayList \(Java Platform SE 7\)](#)

*java.lang.Object extended by java.util.AbstractCollection<E> extended by ...
download.oracle.com/javase/7/.../java/.../ArrayList.html - [En caché](#) - [Similares](#)*

Nosotros en este curso trabajaremos con Java Platform SE 6 (Standard Edition) o Java SE 7 por ser las versiones más usadas hoy en día: si miramos la documentación correspondiente a versiones anteriores podemos confundirnos. Los ejemplos que mostramos en el curso son de Java SE 7. Por tanto una búsqueda más correcta sería “ArrayList api java 7”, y en todo caso **estar atentos a la especificación de la documentación** para comprobar que efectivamente se corresponde con la versión con la que estemos trabajando. Si quieras utilizar otra versión Java no hay problema siempre que sea versión 6 o superior. Los cambios entre versiones no suelen ser tan importantes como para afectar a una persona que aprende el lenguaje por primera vez: en realidad nos daría igual usar una versión u otra. Sin embargo, hay que tener claro qué versión es la que usamos.

Hemos usado el término **api** en las búsquedas: estas siglas corresponden a “Application Programming Interface” o interfaz de programación de aplicaciones. De momento, pensar que API equivale a “recursos” que nos ofrece el lenguaje Java (o si se prefiere, recursos que nos ofrece la empresa que lo desarrolla) para crear aplicaciones. Por ejemplo, podemos pretender ordenar una lista de números denominada Lista1. Podemos hacerlo de dos maneras: escribir las instrucciones paso a paso para que tenga lugar la ordenación, o usar un recurso ya disponible (algo así como “Lista1.usarRecursoOrdenar”). A medida que vayamos avanzando, nos iremos familiarizando poco a poco con el API de Java.

Próxima entrega: CU00607B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ SON LAS ACTUALIZACIONES JAVA? EVOLUCIÓN DEL JAVA DEVELOPMENT KIT JDK

Hemos dicho que existen distintas distribuciones de Java (como J2ME, J2EE, J2SE / Java SE) y distintas versiones (para el Java standard desde JDK 1.0, hasta J2SE 5.0, Java SE 7, Java SE 8, etc.).



Pues bien, **dentro de cada versión existen “actualizaciones” periódicas**, digamos que pequeños cambios o mejoras que la empresa desarrolladora va publicando cada cierto tiempo. El objetivo de estas actualizaciones suele ser corregir pequeños errores o problemas de seguridad a medida que se van detectando. Por ejemplo, para Java SE 7, ha habido numerosas actualizaciones. Si nos fijamos en el software de desarrollo Java más usado, denominado “Java Development Kit” o JDK, para Java SE 7 ha habido numerosas versiones:

1. JDK 1.7.0_01: fue el software inicial o primera actualización.
2. JDK 1.7.0_02: introdujo algunas mejoras. Fue la segunda actualización.
3. JDK 1.7.0_03, JDK 1.7.0_04, JDK 1.7.0_05 ... hasta JDK 1.7.0_51, etc. han sido nuevas actualizaciones que fueron surgiendo con el paso del tiempo.

Cuando instalamos Java en nuestro ordenador, hemos de elegir entre una maraña de posibilidades. Distintos paquetes con distintas configuraciones, y además distintas subversiones o actualizaciones. ¿Cuál elegir? Cualquiera dentro de las más recientes. Nuestra opinión consiste en que es preferible dejar esa carrera loca de las actualizaciones para los programadores profesionales y empresas que lo requieran. En cambio, los usuarios normales o personas que están aprendiendo el lenguaje podemos contentarnos con programas que funcionen, aunque no sean “lo último”. Por tanto, respecto a qué versión de Java usar, usa la que quieras. Sigue nuestras recomendaciones y no te preocupes por las actualizaciones.

Una vez tengas instalado Java, es posible que se te actualice cada cierto tiempo en segundo plano, o bien que te pregunte siquieres actualizar cada cierto tiempo. Nosotros muchas veces desactivamos las actualizaciones porque nos resulta un poco molesto que cada pocos días nos esté pidiendo actualizar.

Próxima entrega: CU00608B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

DESCARGAR (DOWNLOAD JAVA). OBTENER EL INSTALADOR.

Si no tienes instalado Java en tu ordenador, es necesario que lo instales para poder comenzar a programar. Java, aparte de ser el nombre del lenguaje, es también el nombre del programa o conjunto de programas que necesitamos para poder escribir código en nuestro ordenador. Si no estás seguro de si tienes Java instalado puedes chequearlo de dos maneras:

- Vete al directorio donde se encuentran instalados los programas, normalmente C:/Archivos de programa (C:/Program Files) y comprueba si existe una carpeta que se llame Java. Si es así ábrela y comprueba si existe un directorio de nombre similar a jdk1.6.0_xx ó jdk1.7.0_xx ó jdk1.8.0_xx como se ve en la imagen.



Si es así ya tienes instalado Java y puedes omitir los pasos de instalación que describimos más adelante.

- En la dirección de internet <http://www.java.com/es/download/installed.jsp> puedes chequear si tienes instalado Java y qué versión. Te ofrecerá la posibilidad de actualizar si detecta una versión que no sea la última disponible.

Para instalar Java el primer paso será ir a la dirección <http://Java.sun.com/j2se/>. Esta es una web de la multinacional Oracle (antes era de Sun Microsystems), desarrolladora de Java. Desde ahí bajaremos el Java Development Kit (JDK), que es el entorno Java que utilizaremos para realizar nuestros primeros programas. JDK es un paquete con herramientas, utilidades, documentación y ejemplos para desarrollar aplicaciones Java utilizado por los programadores. La actualización que utilicemos no tiene demasiada importancia. Supongamos que usamos JDK 1.7.0_51. Esta versión no es más que un archivo ejecutable, es decir, que tiene la extensión .exe. En nuestro caso el archivo se llama "jdk-7u51-windows-i586.exe". Fíjate que nos bajamos una versión para Windows, si usas otro sistema operativo tendrás que tenerlo en cuenta. Los pasos para la descarga los iremos explicando a continuación.

Ten en cuenta que el lenguaje de programación **Java tiene versiones para instalarse en la mayoría de los sistemas operativos presentes en el mercado (Windows, Linux, Unix y otros más)**. En este curso usaremos Windows como sistema operativo de referencia, pero ten en cuenta que los pasos serán similares para cualquier otro sistema operativo.

Una vez en la página de la empresa Oracle, que se muestra en el navegador al poner la dirección URL <http://Java.sun.com/j2se/>, lo primero que haremos es posicionarnos con el ratón en el enlace "Downloads", que nos mostrará un menú con varias opciones. Seleccionaremos Java for developers – Java SE Downloads. Es decir, nos vamos a descargar Java Standard Edition o Edición Standard de Java.

Descargar (download) Java. Obtener el instalador de Oracle.



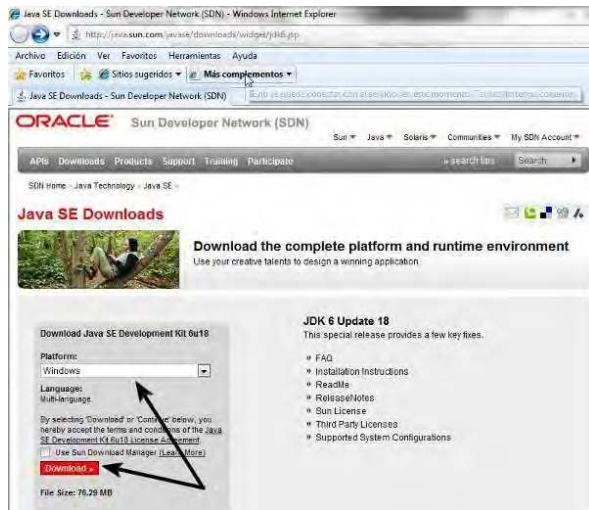
Ten en cuenta que el aspecto de la página de descarga varía cada pocos meses, no te preocupes por no encontrarla con un aspecto determinado, únicamente busca los enlaces como te vamos indicando. Buscamos ahora el enlace “Download JDK 7” ó “Download JDK 8” según la versión que queramos descargar (nosotros usaremos la 7).



A continuación debemos descargar la versión adecuada para nuestro sistema operativo. Para ello tenemos en cuenta que existen múltiples opciones y que tú debes elegir la que se corresponda con el sistema operativo que estés utilizando en tu computador. Si utilizas Windows de 64 bits debes elegir la opción Windows x64. Si utilizas Windows de 32 bits debes elegir la opción Windowsx86. También hay opciones de descarga para otros sistemas operativos (Linux de 32 ó 64 bits, Macintosh, etc.). Si tienes dudas escribe una consulta en los foros aprenderaprogramar.com (<http://aprenderaprogramar.com/foros>).

Ten en cuenta que lo que nos interesa descargar es Java SE Developmet Kit (JDK) y el enlace de descarga tendrá un nombre similar a jdk-7u51-windows-x64.exe si usamos Windows, o terminado en otra extensión si usamos otro sistema operativo como Mac Os ó Linux.

Descargar (download) Java. Obtener el instalador de Oracle.



Si nos pide nombre de usuario (User Name) y Password (contraseña) pulsaremos en "Skip this Step" (saltar este paso) ya que no es obligatorio el registro como usuarios.

Haremos click con el ratón en el enlace indicado con el nombre de archivo (que será del tipo jdk... .exe si es para Windows), para empezar la descarga del instalador Java. En nuestro caso pulsamos sobre el link para descargar el archivo "jdk-7u51-windows-x64.exe" y lo guardamos en nuestro disco duro. El archivo puede ocupar bastante (más de 120 Mb), por lo que necesitamos una buena conexión a internet si queremos que la descarga sea rápida.

Con esto hemos finalizado la descarga del instalador de Java. Si tienes problemas para descargar desde internet, también puedes conseguir el instalador Java en cd's de revistas especializadas o de libros que están disponibles en bibliotecas públicas. Una vez descargado, el siguiente paso será instalar Java en nuestro ordenador.

Próxima entrega: CU00609B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

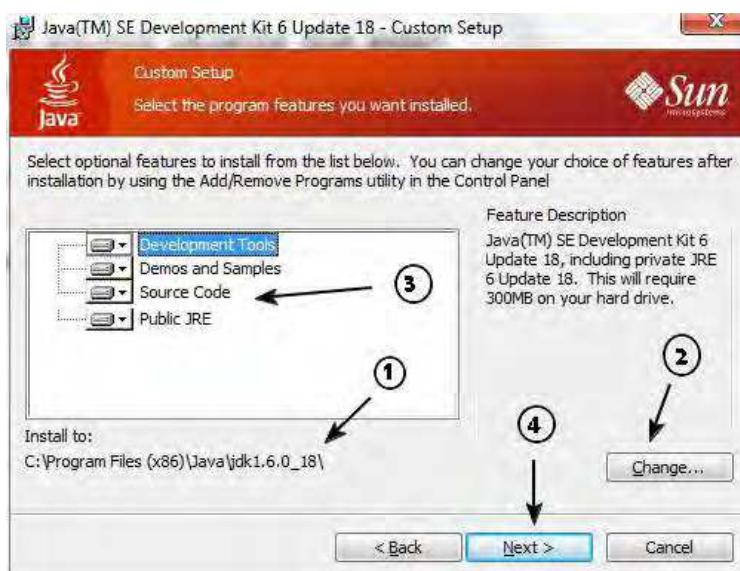
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

INSTALAR JAVA EN NUESTRO PC BAJO WINDOWS

De cara a la instalación de Java con el Sistema Operativo Windows **puede haber pequeñas diferencias según la versión que usemos** (Windows 8, Windows 7, Windows Vista, etc.). Primero, tenemos que hacer doble click sobre el archivo instalador de Java, que habremos descargado anteriormente de nombre jdk-7u51-windows-i586.exe o similar.



Aparecerá una ventana inicial de instalación, informándonos sobre las condiciones de la licencia. Una vez leída, continuamos la instalación pulsando en el botón “Accept” (aceptar). En la ventana que aparece a continuación, lo primero en que debemos fijarnos es en la ruta de instalación (“Install to:”). Es la dirección en la que se instalará Java. Nos aparece la ruta por defecto. Si no estamos de acuerdo con dicha ruta podemos cambiarla con el botón “Change” (cambiar). Como recomendación, aconsejamos evitar cambiarla si no tenemos conocimientos avanzados. El motivo para ello es que muchos programas que trabajan con Java reconocen por defecto dicha ruta. Manteniéndola evitamos problemas y tener que estar configurando la ruta a emplear por el resto de programas.



En la parte central nos señala opciones respecto a qué podemos instalar (Development Tools o herramientas de desarrollo, Demos and samples o ejemplos de prueba, Source Code o código fuente, y Public JRE o conjunto de utilidades Java). Podemos elegir instalar ciertas partes de Java y dejar sin instalar otras. Para ahorrar espacio en el disco duro, podemos no instalar ciertas partes de Java (como los ejemplos de prueba), lo que haríamos deseleccionando esta opción. Si no tienes problemas de espacio lo recomendable es dejarlo todo tal como está y que se instalen todas las opciones. Dicho lo anterior hacemos click en el botón “Next” (siguiente) y continuamos con la instalación.

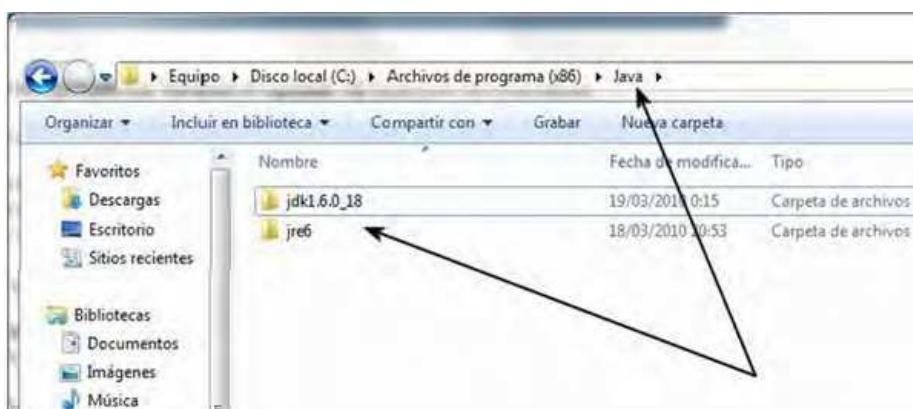
Sin haber terminado el proceso de la ventana anterior, se muestra la siguiente ventana, que es algo parecida a la que vimos anteriormente, donde nos muestra una ruta por defecto en la que se instalará el entorno de Java. El entorno se reconoce por el nombre de “jre7”, presente al final de la ruta.

Siguiendo el mismo criterio comentado anteriormente, es una ruta ya reconocida por defecto por muchos programas que requieren de la “Máquina Virtual” Java. Para evitar problemas de configuración de ruta, es preferible que los usuarios no avanzados no hagan cambios. Si aún así quisiéramos hacerlo, usaríamos el botón “Change...” (cambiar). Seguimos con la instalación, haciendo click en el botón “Next” (siguiente).



Finalmente, si todo ha ido bien, aparecerá una ventana indicando que la instalación se ha completado. Hacemos click en “Finish” para finalizar la instalación.

Ahora vamos a **verificar la instalación**. Para ello vamos a comprobar que se hayan creado dos carpetas con un nombre similar a “jdk1.7.0_51” y “jre7” (ó jdk1.8.0 y jre8) en una ruta similar a “C:\Archivos de programa\Java” (“C:\Program Files\Java”). La primera carpeta, de nombre similar a “jdk1.7.0_51”, corresponde al compilador e intérprete Java, cuyas funciones explicaremos más adelante. La segunda carpeta, de nombre similar a “jre7”, incluye la máquina virtual Java, concepto que también comentaremos más adelante. Si hemos llegado hasta aquí, hemos finalizado correctamente la instalación de Java en Windows.



Próxima entrega: CU00610B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONFIGURAR JAVA EN WINDOWS: VARIABLES DE ENTORNO JAVA_HOME Y PATH.

Java requiere una pequeña configuración para poder usarlo en nuestro equipo. Ten en cuenta que puede haber pequeñas diferencias según la versión que usemos (Windows 8, Windows 7, Windows Vista, etc.). Vamos a ver paso a paso cómo configurar las variables de entorno del sistema necesarias para poder ejecutar Java.

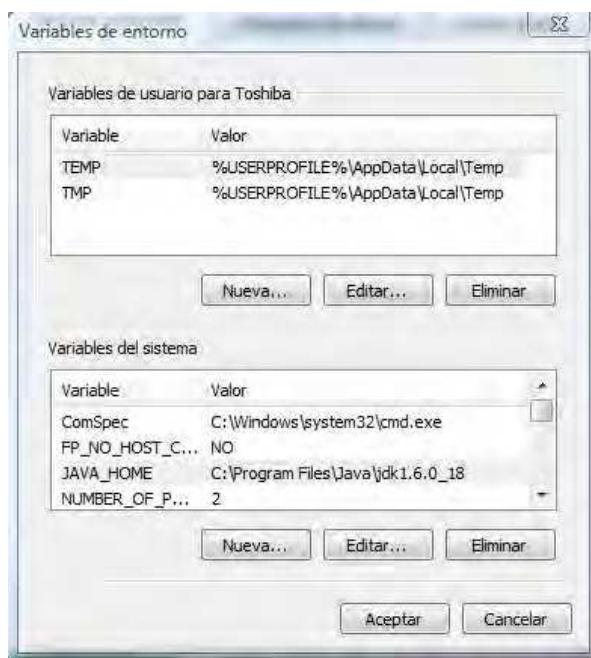


Existen una gran variedad de variables de entorno para diferentes propósitos, entre ellos la comunicación de Java con el sistema operativo. Nos vamos a centrar en sólo dos, las más importantes para nuestra configuración. Estas son las variables que informan al Sistema Operativo dónde y cómo ubicar Java dentro del mismo. Estas variables son: "**JAVA_HOME**" y "**PATH**".

PASO 1: CONFIGURAR LA VARIABLE JAVA_HOME

JAVA_HOME, es una variable de entorno del sistema que informa al sistema operativo sobre la ruta donde se encuentra instalado Java. Seguiremos la siguiente secuencia de pasos para configurar esta variable:

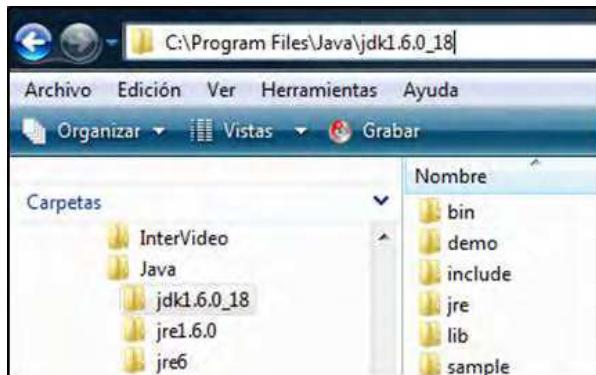
Abrimos el explorador de Windows o pulsamos sobre "Mi Pc". Pulsamos sobre Equipo y con botón derecho del ratón o buscando el ícono → Propiedades → Configuración avanzada / Cambiar configuración → Opciones avanzadas → Variables de entorno → Nueva (Variables del sistema).



Escribiremos en las cajas de texto que se muestran lo siguiente:

Nombre de variable : JAVA_HOME

Valor de variable : escribiremos aquí la ruta en que se haya instalado Java. Puedes consultarla en el propio explorador de Windows buscando la carpeta en que se ha instalado Java, que normalmente será del tipo C:\Program Files\Java\jdk1.7.0_51 ó C:\Program Files (x86)\Java\jdk1.7.0_51. Fíjate en la barra superior donde aparece la ruta y cópiala tal y como aparece ahí.



PASO 2: CONFIGURAR LA VARIABLE PATH

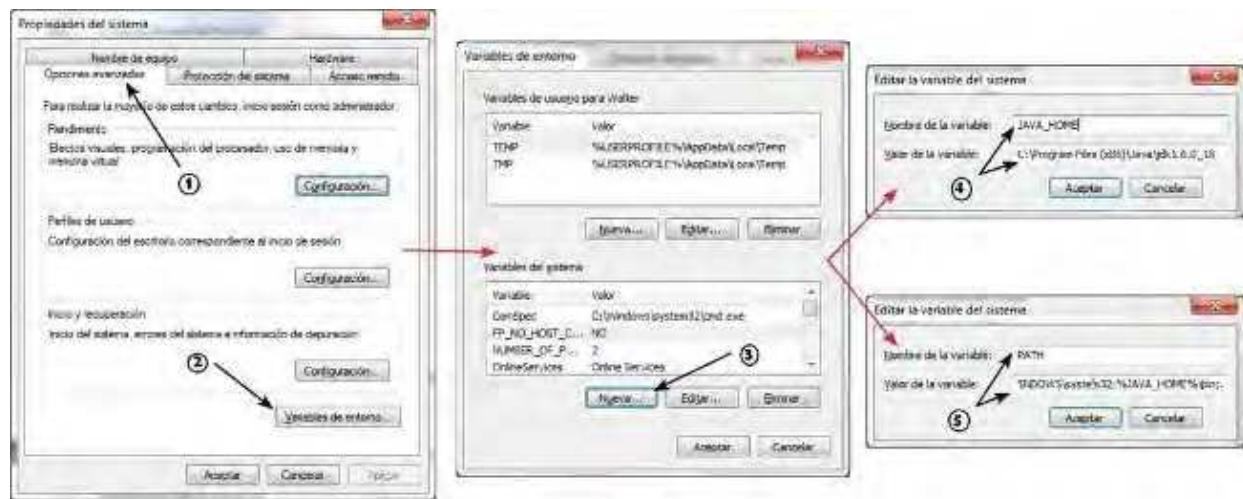
PATH es una variable de entorno del sistema que informa al sistema operativo sobre la ruta de distintos directorios esenciales para el funcionamiento del ordenador. Vamos a añadir al contenido de la variable PATH el lugar donde se encuentran los ficheros ejecutables de Java necesarios para su ejecución, como el compilador (javac.exe) y el intérprete (java.exe). Seguiremos la siguiente secuencia de pasos para configurar esta variable.

Abrimos el explorador de Windows o pulsamos sobre “Mi Pc”. Pulsamos sobre Equipo → Propiedades y con botón derecho del ratón o buscando el ícono → Configuración avanzada / Cambiar configuración → Opciones avanzadas → Variables de entorno → Editar (Variables del sistema). Luego al final del contenido que ya exista, añadiremos un punto y coma y el texto %JAVA_HOME%\bin. No deben quedar espacios intermedios. Nos quedará similar a esto:

Nombre de variable : PATH

Valor de variable : C:\WINDOWS;C:\WINDOWS\system32;%JAVA_HOME%\bin

A modo de resumen de todo el proceso de configuración: hemos creado una variable de entorno llamada JAVA_HOME y hemos añadido una expresión a la variable PATH.



Asegúrate de que todo ha ido bien cerrando todas las ventanas y entrando a “Variables de entorno” para comprobar que aparece todo como hemos indicado. Si es así, ya tenemos Java instalado y configurado en nuestro ordenador.

Próxima entrega: CU00611B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

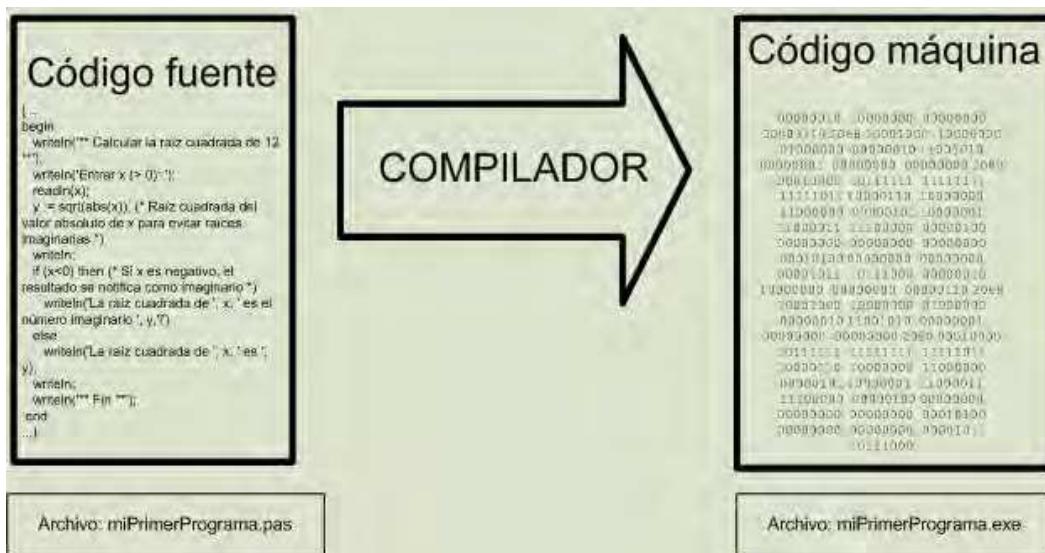
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

MÁQUINA VIRTUAL JAVA (JAVA VIRTUAL MACHINE O JVM). COMPILADOR E INTÉRPRETE. BYTECODE.

Vamos a crear nuestro primer programa, que nos servirá para comprobar si hemos instalado y configurado correctamente Java. Pero antes vamos a repasar algunos conceptos importantes que nos permitan entender lo que vamos haciendo.

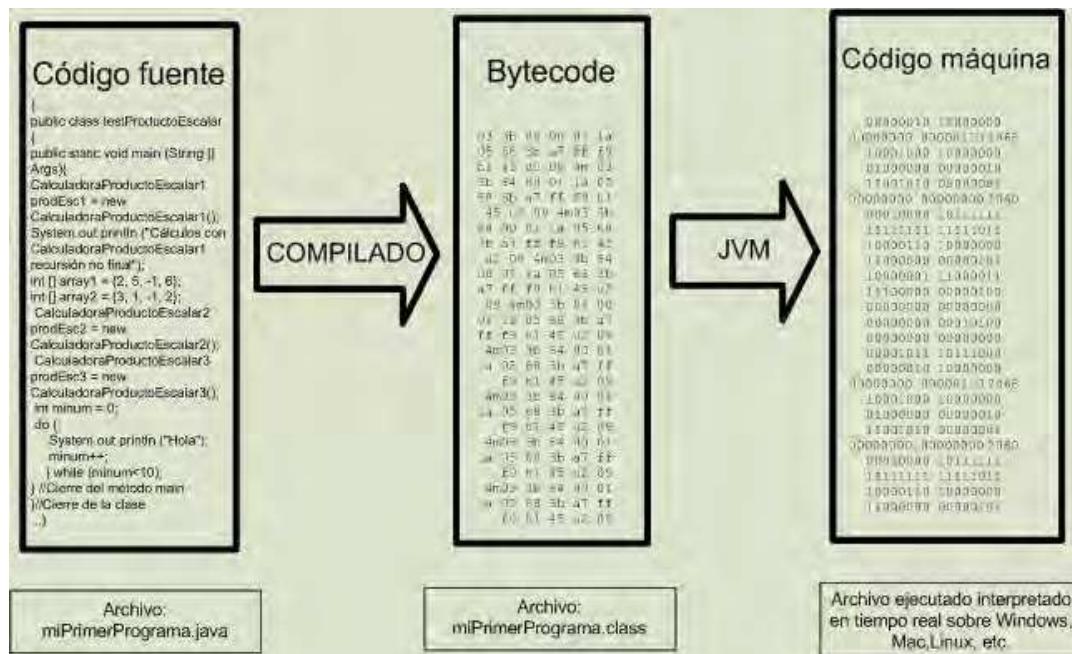


El primer concepto a abordar es el de compilación. “Compilar” significa traducir el código escrito en “Lenguaje entendible por humanos” (por ejemplo Java, C, Pascal, Fortran), a un código en “Lenguaje Máquina”, que entienden las máquinas, pero no entendible por nosotros. Se hace esto porque a los humanos nos resultaría casi imposible trabajar directamente con el lenguaje de los ordenadores. Es por eso por lo que usamos un lenguaje más asequible para nosotros (en nuestro caso Java) y luego empleamos un traductor (compilador). La creación de programas en muchos lenguajes se basa en el proceso: escribir código fuente → compilar y obtener programa ejecutable. El compilador se encarga de evitar que se pueda traducir un programa con código fuente mal escrito y de hacer otras verificaciones previas, de modo que el código máquina tiene ciertas garantías de que cumple cuando menos con los estándares de sintaxis obligatorios de un lenguaje.

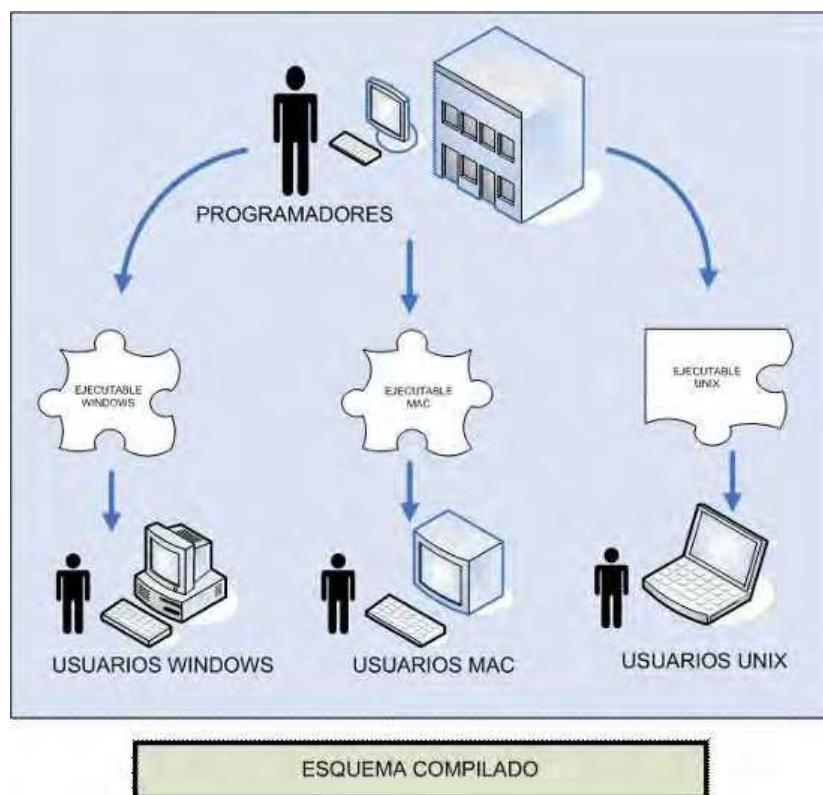


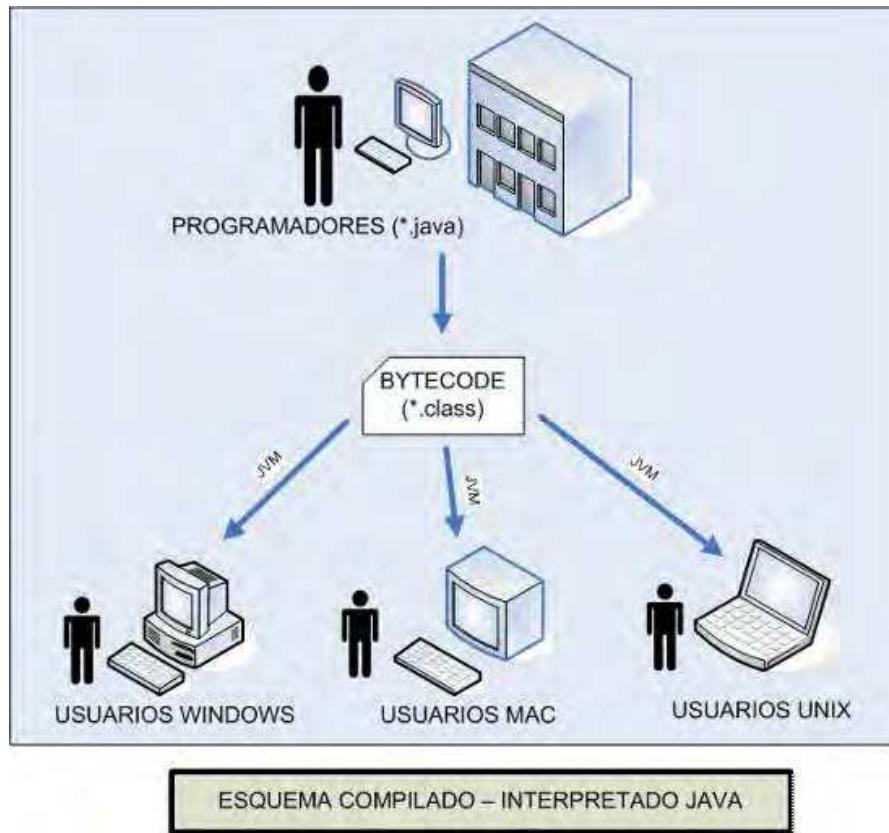
En este esquema, el archivo ejecutable no es válido para cualquier ordenador. Por ejemplo, si se ha generado el ejecutable para Windows, no podrá utilizarse en Macintosh. Sin embargo el proceso en Java no se corresponde con el gráfico anterior. Esta fue una característica novedosa de Java respecto a otros lenguajes cuando se lanzó la primera versión de Java. La novedad introducida fue que **Java se hizo independiente del hardware y del sistema operativo en que se ejecutaba**. En otros lenguajes existía el problema de compatibilidad descrito. Sin embargo, Java se hizo independiente de la plataforma añadiendo un paso intermedio: los programas Java no se ejecutan en nuestra máquina real (en nuestro ordenador o servidor) sino que Java simula una “máquina virtual” con su propio hardware y sistema operativo. En resumen, el proceso se amplía en un paso: del código fuente, se pasa a un código

intermedio denominado habitualmente “bytecode” entendible por la máquina virtual Java. Y es esta máquina virtual simulada, denominada Java Virtual Machine o JVM, la encargada de interpretar el bytecode dando lugar a la ejecución del programa.



Esto permite que Java pueda ejecutarse en una máquina con el Sistema Operativo Unix, Windows, Linux o cualquier otro, porque **en realidad no va a ejecutarse en ninguno de los sistemas operativos, sino en su propia máquina virtual** que se instala cuando se instala Java. El precio a pagar o desventaja de este esquema es que todo ordenador que quiera correr una aplicación Java ha de tener instalado Java con su máquina virtual. Las diferencias entre ambas concepciones podemos verlas en los siguientes esquemas.





La máquina virtual era un aspecto importante que diferenciaba a Java de otros lenguajes cuando irrumpió en el mercado de los lenguajes de programación; permitía **escribir y compilar el programa una sola vez** en lugar de varias veces y ejecutar ese código en cualquier plataforma (“write once, run anywhere”).

Otra razón de su gran éxito ha sido que cuando surgió se convirtió en un lenguaje más orientado a objetos que todos los otros lenguajes existentes. Además cabe destacar su potencia y el permitir crear programas de aspecto y funcionamiento muy similar al también muy popular “entorno Windows”. Esto afianzó su reconocimiento como un lenguaje de programación innovador.

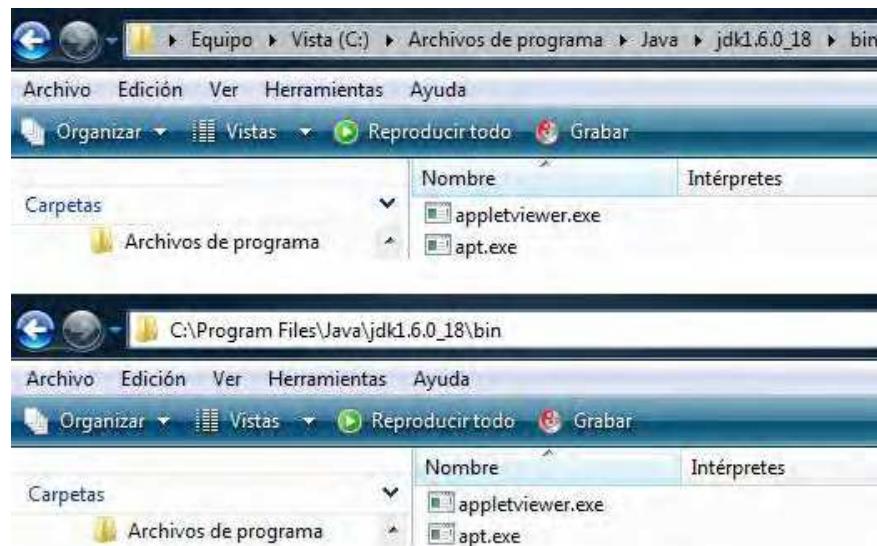
Aclarar que al ser Java un programa que se interpreta en una máquina virtual, el archivo resultante de la compilación es un archivo con la extensión .class interpretable por la máquina virtual. Este archivo .class está escrito en un *lenguaje de máquina virtual* (bytecode).

Para que la “Máquina Real” (nuestro ordenador) ejecute el programa, hay que “interpretar” (traducir) el archivo .class a un código en “Lenguaje de Máquina Real”. Esta es la labor de lo que llamamos “intérprete” o traductor del lenguaje de la máquina virtual a la máquina real.

Los archivos respectivos que se encargan de estas tareas son:

- El compilador Java --- > `javac.exe`. Se encarga de compilar el código fuente.
 - El intérprete Java --- > `java.exe`. Se encarga de interpretar los archivos `.class` (bytecode).

La ruta en la que se ubican ambos archivos es esta o una similar a esta: "C:\Program Files (x86)\Java\jdk1.7.0_51\bin" (o "C:\Program Files\Java\jdk1.7.0_51\bin", depende de la versión de Windows en caso de que usemos Windows). El explorador de Windows nos muestra una barra con la ruta en que nos encontramos (ruta aparente). Para conocer la ruta real basta pinchar sobre la ruta aparente.



En la próxima entrega veremos los pasos para compilar e interpretar nuestro primer programa escrito en lenguaje Java.

Próxima entrega: CU00612B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

COMPILAR E INTERPRETAR NUESTRO PRIMER PROGRAMA

Veamos los pasos para compilar e interpretar nuestro primer programa escrito en lenguaje Java.



PASO 1: Creación del código fuente

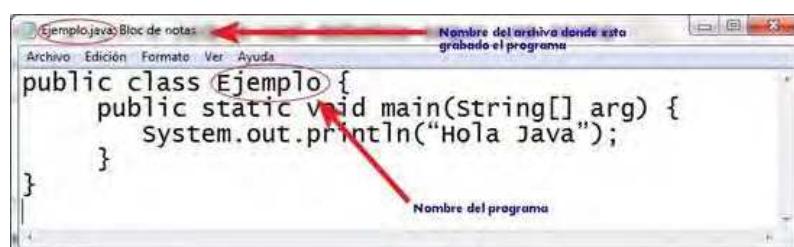
Abrimos el Bloc de notas de Windows (Inicio -> Todos los programas -> Accesorios -> Bloc de notas), que usaremos ahora como editor de trabajo por su simplicidad. Más adelante veremos un editor más sofisticado. Si prefieres usar otro editor en este momento no hay problema (por ejemplo WordPad, Notepad, etc.). Da igual mientras que se grabe el contenido como “texto sin formato” (en código ASCII). Una vez que tengamos el bloc de notas abierto escribiremos nuestro primer programa, que mostrará un texto “Hola Java” en la ventana consola DOS (ventana del sistema que más adelante explicaremos cómo manejar). Aclarar que por el momento no debemos preocuparnos de entender lo que escribimos: esto lo explicaremos más adelante. Nuestro objetivo ahora es simplemente comprobar que podemos ejecutar un programa escrito en Java.

El código de nuestro programa en Java, escrito en el bloc de notas, será el siguiente:

```
public class Ejemplo {
    public static void main(String[] arg) {
        System.out.println("Hola Java");
    }
}
```

Los caracteres de llaves y corchetes se escriben pulsando ALT GR + la tecla correspondiente.

A continuación procedemos a grabar nuestro programa: pulsamos en Archivo -> Guardar como y le ponemos como nombre Ejemplo.java. Estamos siguiendo una norma dictada por Java, que es el nombre del archivo (Ejemplo.java) y el nombre del programa (que hemos definido dentro del archivo después de escribir public class, y que también es “Ejemplo”) sean idénticos. Resumidamente: hemos de hacer coincidir nombre del archivo y nombre del programa, tanto en mayúsculas como en minúsculas, y la extensión del archivo habrá de ser siempre “.java”.



Para seguir un orden y evitar problemas posteriores durante la compilación, haremos lo siguiente. Crearemos una carpeta en C: denominada “Ejercicios” y ubicaremos el programa Ejemplo.java dentro de esta carpeta.

PASO 2: Compilación y ejecución del programa

Vamos a proceder a la compilación e interpretación de este pequeño programa Java. Lo haremos usando la ventana consola del DOS. Para visualizar esta ventana debemos proceder así: Inicio -> Todos los programas -> Accesorios -> Símbolo del sistema. Sobre la ventana consola escribiremos el comando “cd C:\Ejercicios” y pulsaremos enter. Esto nos traslada a la carpeta donde se ubica nuestro archivo (“Ejemplo.java”). Si hemos accedido correctamente al escribir dir y pulsar enter nos debe aparecer el archivo Ejemplo.java. A continuación daremos la instrucción para que se realice **el proceso de compilación del programa**, para lo que escribiremos “javac Ejemplo.java”, donde “javac” es el nombre del compilador (javac.exe) que transformará el programa que hemos escrito nosotros en lenguaje Java al lenguaje de la máquina virtual Java (bytecode), dando como resultado un nuevo archivo “Ejemplo.class” que se creará en este mismo directorio. Si te aparece un mensaje de error revisa todos los pasos anteriores que hemos explicado, pues es posible que no se hayan creado las variables de entorno o que el archivo no esté bien ubicado, etc. Para comprobar si se ha creado el archivo escribiremos en la ventana consola el comando “dir”. Comprobaremos que nos aparecen dos archivos: Ejemplo.class (bytecode creado por el compilador) y Ejemplo.java (código fuente creado por nosotros).

Finalmente, vamos a pedirle al intérprete que ejecute el programa, es decir, que transforme el código de la máquina virtual Java en código máquina interpretable por nuestro ordenador y lo ejecute. Para ello escribiremos en la ventana consola: java Ejemplo.

El resultado será que se nos muestra la cadena “Hola Java”. Si logramos visualizar este texto en pantalla, ya hemos desarrollado nuestro primer programa en Java.

```

Símbolo del sistema
Microsoft Windows [Versión 6.1.7601]
Copyright © 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\Walter>cd c:\Ejercicios
C:\Ejercicios>javac Ejemplo.java
C:\Ejercicios>java Ejemplo
Hola Java
C:\Ejercicios>

```

En otros sistemas operativos el proceso es similar. Si tienes problemas para completar este ejemplo no te preocupes, sigue avanzando con los contenidos del curso.

Próxima entrega: CU00613B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

NETBEANS, ECLIPSE, JCREATOR, JBUILDER... ¿CUÁL ES EL MEJOR ENTORNO DE DESARROLLO (IDE) PARA JAVA?

Hemos generado nuestro primer programa Java usando las herramientas más básicas posibles: el bloc de notas y la ventana consola de DOS. **Los programadores utilizan herramientas más sofisticadas** ya que facilitan el trabajo enormemente. Dentro de estas herramientas podríamos hablar de entornos de desarrollo (IDEs) o frameworks.



A veces ambos términos se confunden. Nosotros nos referiremos a IDE como a un programa que nos permite desarrollar código en un lenguaje y que incorpora habitualmente:

- a) Un espacio para la escritura de código con cierta ayuda interactiva para generar código y para indicar los errores de sintaxis que se cometan por parte del programador.
- b) La posibilidad de compilar y ejecutar el código escrito.
- c) La posibilidad de organizar los proyectos de programación.
- d) Herramientas auxiliares para programadores para detección de errores o análisis de programas (debuggers).
- e) Otras opciones como utilidades para pruebas, carga de librerías, etc.

Existen diversos IDEs para Java. Vamos a citar algunos de ellos:

- a) **Eclipse:** software libre que se puede descargar en <http://www.eclipse.org>. Es uno de los entornos Java más utilizados a nivel profesional. El paquete básico de Eclipse se puede expandir mediante la instalación de plugins para añadir funcionalidades a medida que se vayan necesitando.
- b) **NetBeans:** software libre que se puede descargar en <https://netbeans.org/>. Otro de los entornos Java muy utilizados, también expandible mediante plugins. Facilita bastante el diseño gráfico asociado a aplicaciones Java.
- c) **BlueJ:** software libre que se puede descargar en <http://bluej.org>. Es un entorno de desarrollo dirigido al aprendizaje de Java (entorno académico) y sin uso a nivel profesional. Es utilizado en distintas universidades para la enseñanza de Java. Destaca por ser sencillo e incluir algunas funcionalidades dirigidas a que las personas que estén aprendiendo tengan mayor facilidad para comprender aspectos clave de la programación orientada a objetos.
- d) **JBuilder:** software comercial. Se pueden obtener versiones de prueba o versiones simplificadas gratuitas en <http://www.embarcadero.com> buscando en la sección de productos y desarrollo de aplicaciones. Permite desarrollos gráficos.

- e) **JCreator:** software comercial. Se pueden obtener versiones de prueba o versiones simplificadas gratuitas en <http://www.jcreator.com>. Este IDE está escrito en C++ y omite herramientas para desarrollos gráficos, lo cual lo hace más rápido y eficiente que otros IDEs.
- f) **Otros.**

¿Qué IDE utilizar? Cada entorno de desarrollo tiene sus ventajas y sus inconvenientes. De cara al aprendizaje puede servirnos cualquiera de estos entornos, aunque nosotros vamos a recomendar y utilizar para este curso BlueJ. El motivo para ello es que es un entorno sencillo. Un manual de BlueJ puede constar habitualmente de 40 o 50 páginas. Un manual para otro entorno puede constar de 400, 4.000 ó 40.000 páginas. Nosotros queremos centrarnos en aprender el lenguaje Java y un entorno sencillo va a facilitar el aprendizaje evitando que nos entretenemos en aprender cuestiones de detalle sobre el IDE. También podemos recomendarlo por incluir algunas herramientas con orientación didáctica para facilitar el aprendizaje. Por supuesto que BlueJ tiene sus inconvenientes como el no ser suficientemente funcional para su uso profesional o el que la ayuda sintáctica contextual no es tan rica como en otros entornos.

¿Desaconsejamos el uso de otro IDE? No, este curso se puede seguir con cualquiera de los IDEs que hemos citado. Hay que tener en cuenta que un IDE es una herramienta y que por tanto podremos usar alternativamente una u otra en función de nuestras necesidades. Pongamos por caso que un IDE fuera un martillo: ¿qué martillo elegir?



Elegiremos como hacemos habitualmente en la vida cotidiana: el que nos recomienda una fuente de confianza, o el que nos resulte más cómodo, o el que mejor sepamos usar. Para cada trabajo o situación que se nos plantee, podremos elegir un tipo de martillo u otro.

BlueJ es un entorno de desarrollo diseñado para el aprendizaje de la programación. No obstante, hay que tener presente que BlueJ es un entorno Java completo. No se trata de una versión de Java simplificada o recortada con fines de enseñanza. Se ejecuta sobre la herramienta profesional para Java de Oracle (antes Sun Microsystems) denominada JDK (Java Development Kit) y utiliza el mismo compilador y máquina virtual que otros entornos como Eclipse o NetBeans. Por tanto BlueJ cumple con la especificación oficial de Java, aunque obviamente es más simplificado y tiene bastantes menos funcionalidades que otros entornos de uso profesional. BlueJ está pensado para que las personas que aprenden programación puedan *aprehender* la filosofía y metodología de la programación orientada a objetos, que es el marco dentro del cual se encuentra Java.

Próxima entrega: CU00614B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

DESCARGAR (DOWNLOAD) EL IDE PARA JAVA BLUEJ. INSTALACIÓN EN WINDOWS.

Para instalar BlueJ es necesario tener instalado Java en nuestro ordenador, ya que sin el compilador y máquina virtual no podremos ejecutar nuestros programas Java.



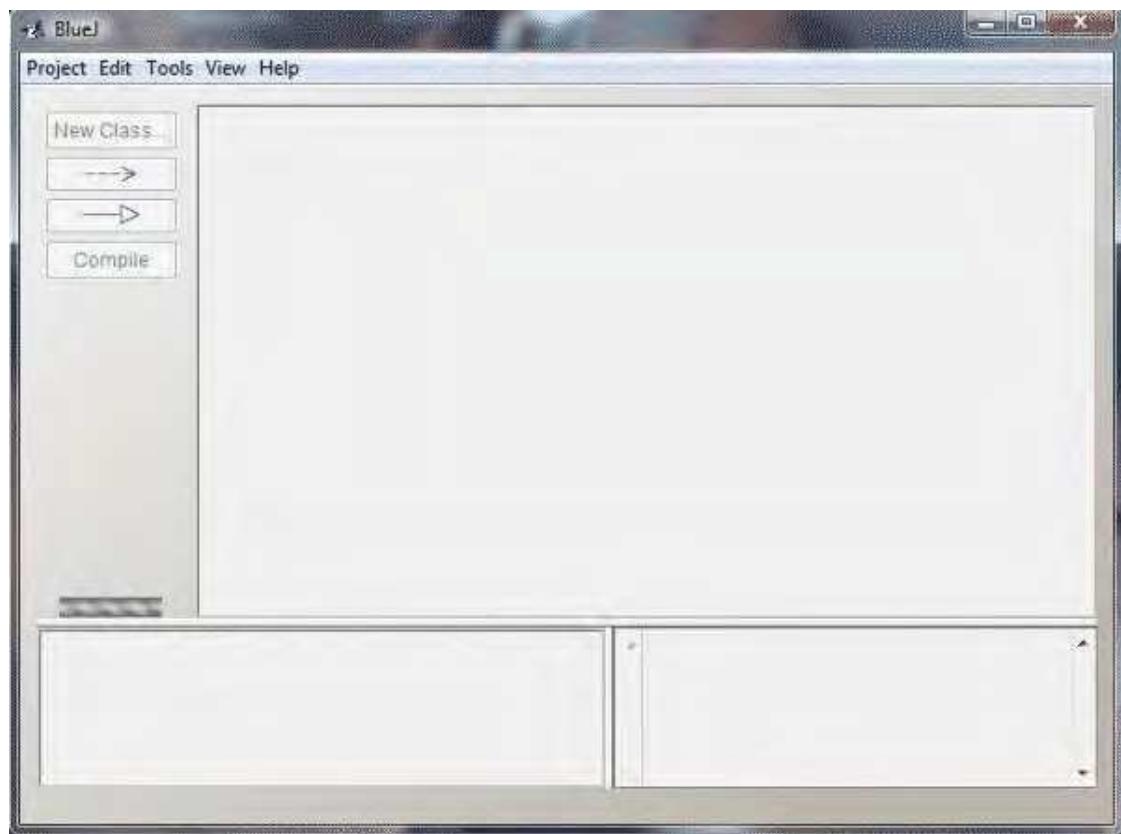
Para instalar BlueJ accederemos a la página <http://bluej.org>. Ten en cuenta que la apariencia de esta página cambia cada pocos meses. Buscamos el enlace “Download BlueJ Installer for Windows”, es decir, lo que nos interesa descargar es BlueJ para el sistema operativo que estemos usando. Si es Mac Os X ó Linux, tendremos que buscar el enlace correspondiente. Si nos da opción a elegir BlueJ Combined Installer (includes JDK) esa opción no nos interesa, ya que nosotros ya tenemos instalado el JDK y únicamente queremos instalar BlueJ.

Una vez pulsamos en el enlace de descarga nos aparecerá un mensaje preguntando si queremos descargar el archivo (tipo blueJsetup-311.exe ó bluej-311.msi si es para Windows, o con otras extensiones si es para otro sistema operativo) en nuestro ordenador.

A screenshot of the BlueJ website. At the top, there's a navigation bar with a blue jay icon, the text "BlueJ – The interactive Java environment", and a "Search the BlueJ site" input field. Below the navigation, there's a sidebar with links: "home", "about BlueJ", "download", "documentation", "extensions", "help & Info", and "resources". The main content area has two sections: "Step 1: Get Java" and "Step 2: Download the installation package". Under "Step 1: Get Java", it says "Check the system requirements (right side of this page). Make sure you have an appropriate version of Java installed." Under "Step 2: Download the installation package", it lists "BlueJ version 3.0.4" with options for different operating systems: "for Windows 2000, XP, Vista or newer (5.5 Mb)" leading to "blueJsetup-304.exe", "for Mac OS X (5.1 Mb)" leading to "BlueJ-304.zip", "for Debian, Ubuntu and other Debian-based systems (5.0 Mb)" leading to "blueJ-304.deb", and "all other systems (executable jar file) (5.1 Mb)" leading to "blueJ-304.jar". At the bottom left of the main content area, there's a link "Discussion Group".

A continuación cerramos el resto de programas y hacemos doble click sobre el fichero descargado, con lo cual comienza la instalación. La instalación es trivial: pulsa aceptar hasta completarla. Si se te presenta algún problema consulta el sitio oficial de BlueJ o en los foros aprenderaprogramar.com. Una vez instalado, BlueJ aparecerá en nuestra lista de programas (Inicio -> Programas -> BlueJ) y tendremos en nuestro escritorio un ícono para acceder al programa.

A través de Inicio -> Programas o mediante el icono, abrimos BlueJ. Si la instalación ha sido correcta nos aparecerá la pantalla de inicio del programa. Si es así, hemos instalado con éxito BlueJ.



Próxima entrega: CU00615B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ ES UN PROYECTO JAVA? ORGANIZACIÓN DE ARCHIVOS .JAVA, .CLASS Y OTROS

Un proyecto Java podemos considerarlo como una serie de carpetas ordenadas y organizadas de acuerdo con **una lógica para mantener organizado el código**. Un proyecto suele constar de archivos .java, archivos .class y documentación.

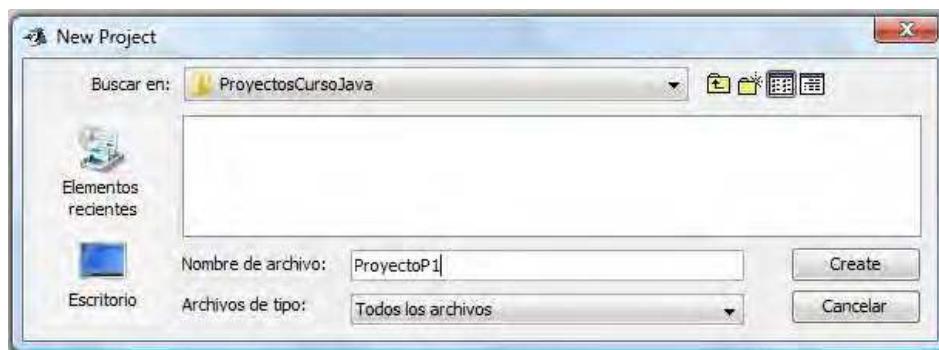


Los archivos .java contienen el código fuente (entendible por humanos) que en un momento dado podemos modificar con un editor de textos y suelen encontrarse en carpetas de nombre src (source). Los archivos .class contienen el bytecode (no entendible por humanos pero sí por la máquina virtual Java) y suelen encontrarse en carpetas de nombre bin (binary).

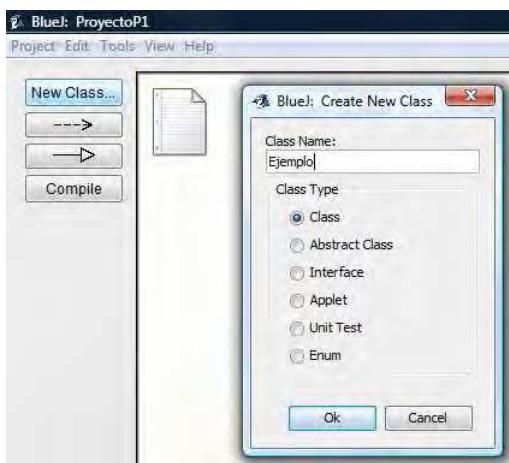
La organización de los archivos en carpetas y la presencia de otros adicionales depende del entorno de desarrollo que utilicemos. Además, Java introduce un esquema organizativo a través de paquetes (packages) que comentaremos más adelante.

Para trabajar con proyectos en la mayoría de entornos, incluido BlueJ, debemos tenerlos en un soporte grabable accesible (por ejemplo en una carpeta de nuestro disco duro). No es válido por tanto un cd, dvd, unidad remota restringida o carpeta del disco duro con restricciones. El motivo es que los entornos de desarrollo trabajan grabando información en la carpeta del proyecto que se encuentre activo. Por tanto un soporte no escribible no es adecuado. Si queremos trabajar con un proyecto contenido en un cd o dvd, primero lo copiaremos a nuestro disco duro y después lo abriremos para trabajar con él.

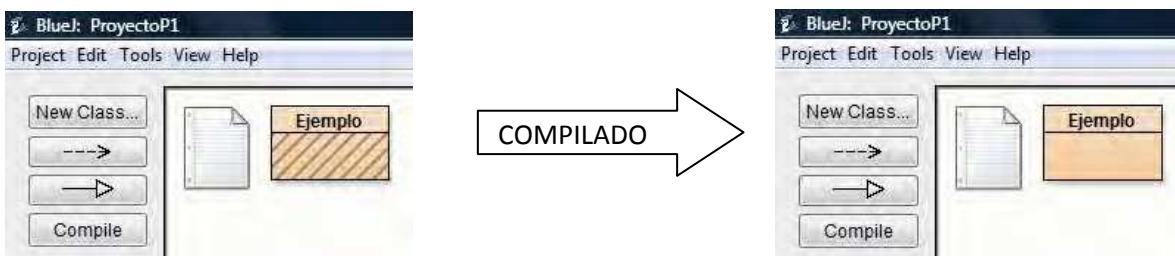
Vamos a crear nuestro primer proyecto. Para ello conviene crear primero una carpeta donde ir almacenando los proyectos que vayamos creando. Hazlo en la ruta que te parezca más adecuada. Nosotros usaremos C:/ProyectosCursoJava. Pulsamos en el menú Project → New Project y buscamos la carpeta donde vamos a guardar el proyecto.



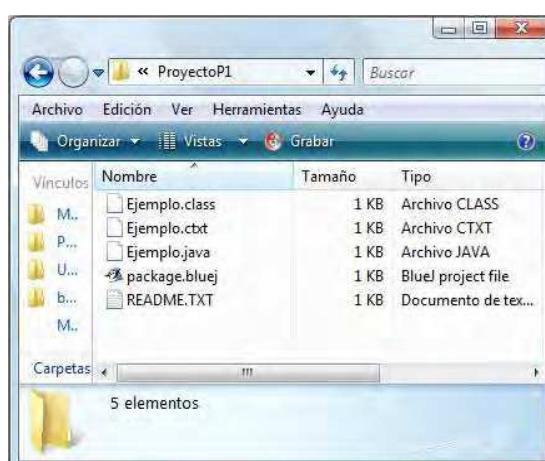
Donde pone “Nombre de archivo” escribiremos ProyectoP1 o cualquier otro nombre que nos parezca adecuado.



A continuación, en el lateral izquierdo pulsamos sobre New Class y donde nos solicita nombre para la clase (Class Name) escribimos “Ejemplo” y pulsamos aceptar. Nos aparecerá un ícono con el nombre “Ejemplo” y rayas transversales. Ese ícono representa una clase. Discutiremos qué es una clase en Java un poco más adelante, por ahora simplemente pensaremos que una clase es código Java. Ahora vamos a ejecutar una pequeña prueba. Pulsa sobre el botón “Compile” y el rayado que aparecía deberá haber desaparecido.



¿Qué hemos hecho en este proceso? Al crear la clase, hemos creado un archivo denominado Ejemplo.java. Al pulsar sobre “Compile”, hemos transformado el código que contenía ese archivo en bytecode y hemos creado el archivo Ejemplo.class. ¿Pero qué código hemos compilado si no hemos escrito nada? Efectivamente, no podríamos compilar algo vacío. La explicación radica en que cuando se crea una clase vacía BlueJ la crea con un código por defecto, digamos que un ejemplo muy básico de código que ya es compilable. Vamos a comprobar lo que hemos dicho sobre los archivos: para ello vamos al explorador de archivos y nos dirigimos a la ruta donde tenemos guardado el proyecto.



En esta ruta comprobamos los archivos de los que consta nuestro proyecto:

- **Ejemplo.java**: el código fuente en lenguaje Java.
- **Ejemplo.class**: el bytecode o código máquina para la máquina virtual Java.
- **Otros archivos**: archivos creados por BlueJ a los que de momento no vamos a prestar atención.

Pulsa ahora sobre el ícono del archivo Ejemplo.java y con el botón derecho del ratón elige “Abrir” para verlo con un editor de texto como el bloc de notas. Al abrirse el archivo podrás ver el código fuente (e incluso podríamos cambiarlo desde aquí si quisieramos). Cierra el editor y trata de repetir el proceso con el ícono del archivo Ejemplo.class. El resultado será que Windows te indica que no sabe cómo abrir ese archivo, o, si logras abrirlo, que te aparezcan una serie de caracteres “raros”. Esto concuerda con lo que habíamos dicho: el código fuente está constituido por texto y podemos verlo y editarlo. El bytecode es un tipo de código máquina, por tanto no podemos editarla directamente porque no es comprensible para nosotros.

Ya hemos visto que el archivo Ejemplo.java contiene un código. Esa es la razón por la que nos ha sido posible compilar anteriormente usando BlueJ. Vuelve a BlueJ y para acceder al código que se encuentra en la clase hacemos doble click en el ícono, con lo cual se nos abrirá la ventana del editor de BlueJ.

Próxima entrega: CU00616B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

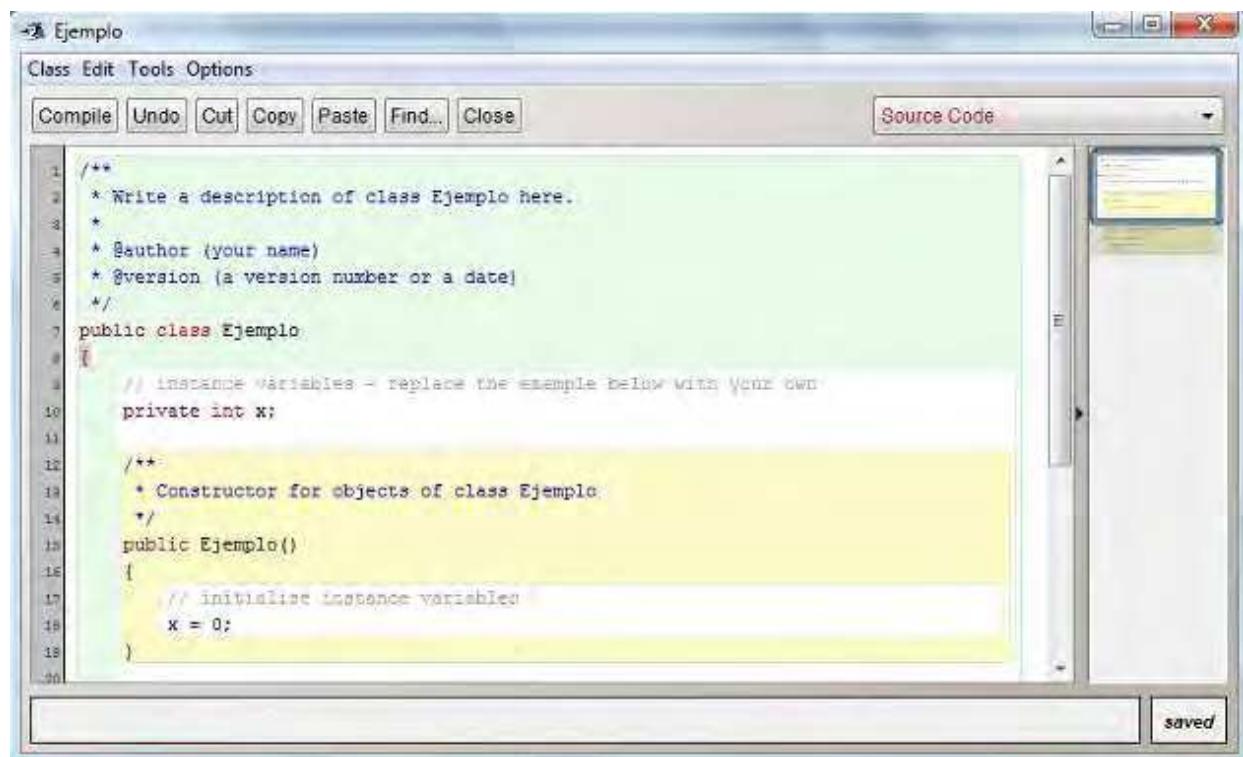
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

LA VENTANA EDITOR EN EL IDE BLUEJ. ESCRIBIR CÓDIGO, COMPILAR, DOCUMENTACIÓN

BlueJ dispone de un editor similar a lo que podría ser el bloc de notas u otros editores como Notepad. Una vez trabajamos con un entorno de desarrollo, normalmente usaremos su editor en lugar de cualquier otro. Podríamos usar más de un editor (por ejemplo el de BlueJ y el bloc de notas, o incluso el de BlueJ y el de Eclipse) pero esto no es recomendable.

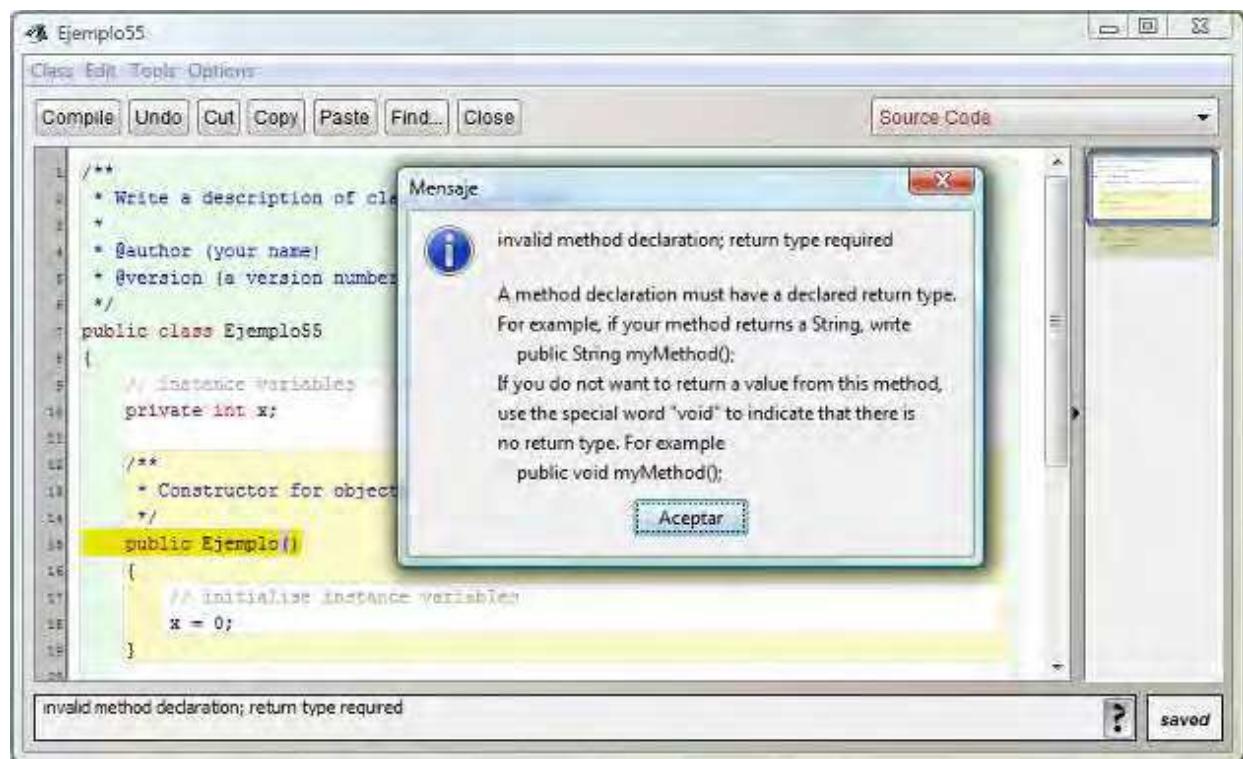


El aspecto del editor de BlueJ es este:



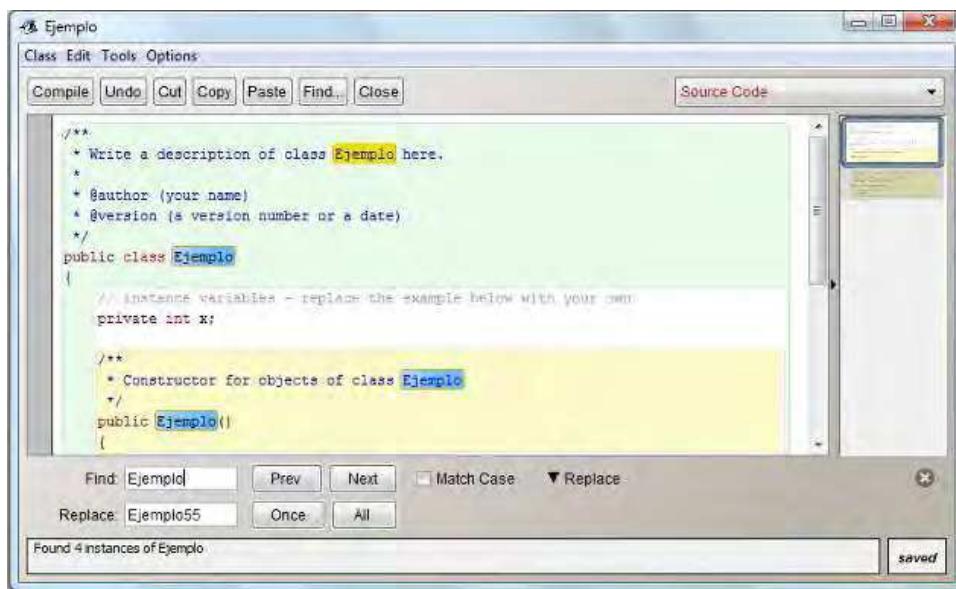
Vamos a fijarnos en algunas cosas de interés:

- a) **El área central** sirve para escribir código.
 - b) **El botón Compile** sirve para compilar el código que se encuentra en la ventana, es decir, crear o actualizar el fichero .class. Si la compilación no es posible se nos informará mediante un mensaje de error. Prueba a forzar un error. Para ello cambia el texto “public class Ejemplo” por “public class Ejemplo55” y pulsa Compile.
 - c) **La zona auxiliar** en la parte inferior: en ella nos aparecerán los mensajes de error de compilación cuando los haya (cosa que será habitual, pues al escribir código es frecuente cometer errores).



Una vez salta un error, nos aparece en la zona de mensajes una notificación breve, por ejemplo “invalid method declaration, return type required”. Si pulsamos en el interrogante situado en la parte inferior derecha, se nos abre una ventana con información ampliada sobre el error. Nos puede ser útil cuando no entendamos por qué se está produciendo el error y necesitemos información adicional. Para corregir el error, elimina el “55” adicional que habíamos escrito y vuelve a pulsar *Compile*.

- d) En la parte derecha del editor nos aparece una columna que es una **vista en miniatura** del código fuente. Nos sirve para saber qué cantidad de código fuente hay escrito (si es mucho o poco) y para saber dónde estamos situados en un momento dado (si en la parte inicial, central o final).
- e) En la parte superior derecha nos aparece un desplegable que nos permite elegir entre “**Source code**” (código fuente) y “**Documentation**” (documentación). Esto nos permite alternar entre ver el código y ver su documentación. Hablaremos de la documentación más adelante, por ahora lo pasaremos por alto.
- f) **Otros botones disponibles** son Undo (deshacer), Cut (cortar), Copy (copiar), Paste (pegar), Find (buscar) y Close (cerrar). Nosotros en general usamos combinaciones de teclas como CTRL+Z para deshacer, CTRL+X para cortar, CTRL+C para copiar y CTRL+V para pegar y cerraremos con el aspa de cierre de ventana por lo que damos poco uso a estos botones. Sin embargo, si queremos señalar la importancia de uno de ellos: el botón Find. Palsa este botón y en la parte inferior te aparecerá el área de búsqueda. Palsa ahora sobre “Replace” (reemplazar) para permitir la búsqueda y reemplazo de texto. Como cadena a buscar pondremos Ejemplo y como cadena de reemplazo Ejemplo55.



Fíjate que en el área de notificaciones nos aparece las veces que se ha detectado la presencia de la cadena: "Found 4 instances of Ejemplo". La cadena localizada activa sobre la que nos encontramos se muestra marcada en amarillo, mientras que el resto de apariciones se muestra en azul. Usando los botones *Prev* y *Next* nos podemos desplazar a una aparición previa o siguiente de la cadena. Pulsando *Once* podemos reemplazar la cadena localizada activa, mientras que pulsando *All* reemplazamos todas las coincidencias. Vamos a pulsar *All* y a compilar otra vez. Comprobamos que la compilación es correcta porque en el área de notificaciones nos aparece el mensaje *Class compiled – No syntax errors*.

Falta por comentar la casilla de opción "**Match Case**". Si está activada la búsqueda sólo devuelve palabras con coincidencia exacta (diferenciando entre mayúsculas y minúsculas), mientras que estando desactivada localiza todas las cadenas aunque no haya coincidencia exacta de mayúsculas y minúsculas.

Esta herramienta de búsqueda y reemplazo es mucho menos potente que otras a las que quizás estemos acostumbrados como la de Microsoft Word, pero es fácil de usar y nos va a resultar suficiente para la creación de nuestros programas en Java.

- g) En la parte superior izquierda nos aparece el icono de BlueJ y el nombre actual de la clase (este nombre puede cambiar cuando compilamos). Inmediatamente debajo aparecen varios menús: Class, Edit, Tools, Options. Estos menús los descubriremos a medida que nos vaya siendo necesario. Vamos a citar ahora dos posibilidades interesantes. Con *Edit -> AutoLayout* podemos auto-ordenar la presentación estética del código. Es decir, después de escribir muchas líneas que quizás nos hayan quedado desalineadas, mal tabuladas, etc. con esta opción se nos alinearán de forma adecuada para su lectura. Ten en cuenta que el código es necesario leerlo con frecuencia, de ahí que el que su aspecto sea claro y ordenado es importante para facilitar la lectura. La otra posibilidad a la que nos referímos es *Option -> Preferences ->Editor -> Display Line Numbers*. Activando esta opción se nos mostrarán **números de línea en una columna en el lateral izquierdo del editor**. Resulta de interés ya que en muchas ocasiones el código consta de miles de líneas y puede ser necesario que nos apuntemos en un papel un cambio que queda pendiente en una línea. Por otro lado, cuando salten errores al compilar o ejecutar muchas veces podremos localizar el error por número de línea.

Próxima entrega: CU00617B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

ESCRIBIR CÓDIGO (UNA CLASE) EN JAVA CON UN IDE. PRIMER PROGRAMA.

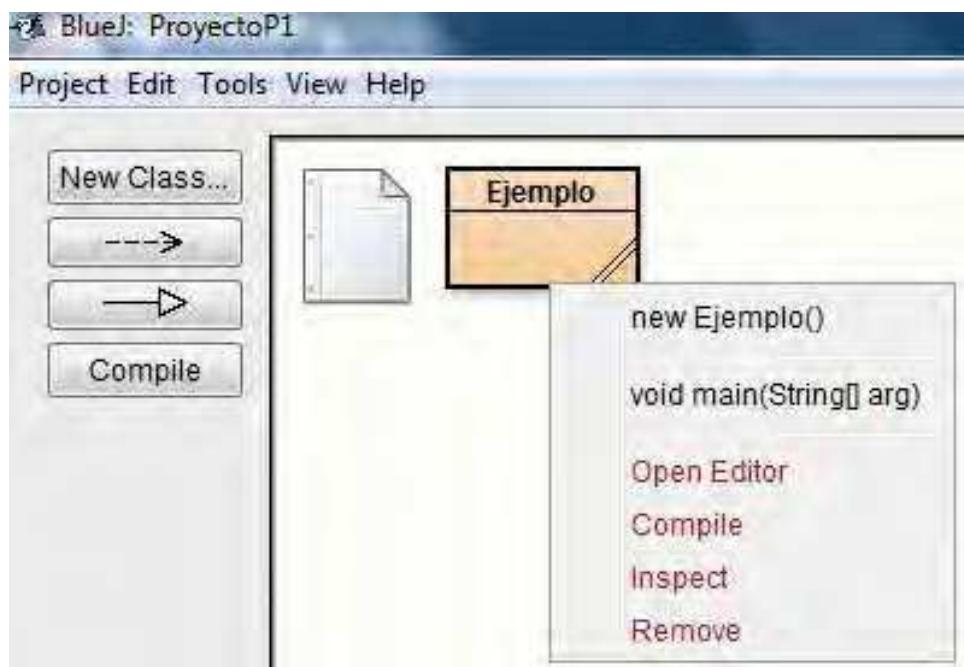
Ya tenemos una clase creada en nuestro entorno de desarrollo. Pero el código existente es un código de ejemplo que, aunque nos puede ser útil en algún momento, en general no nos va a interesar. Nosotros escribiremos nuestro código partiendo de cero. Es la mejor manera de aprender.



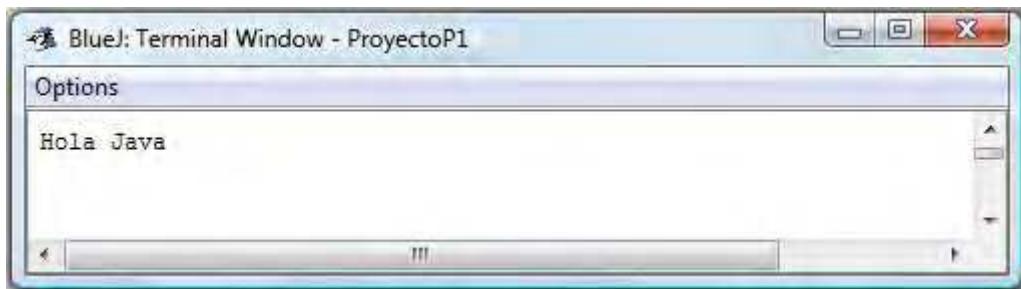
Por ello, abrimos el editor y borramos todo el código existente dejando el editor sin contenido. A continuación escribiremos lo siguiente:

```
public class Ejemplo {  
    public static void main(String[] arg) {  
        System.out.println("Hola java");  
    }  
}
```

Seguidamente hacemos AutoLayout, compilamos el código y cerramos el editor. En la ventana principal de BlueJ veremos el ícono de la clase Ejemplo. Nos situamos sobre él y pulsamos el botón derecho



En el menú desplegable, elegimos la opción void main (String[] arg). Pulsamos OK y nos aparece la ventana de consola de BlueJ. La ventana de consola es la ventana más básica dentro de un entorno de desarrollo: sirve para mostrar mensajes de texto y es la ventana de uso más habitual cuando se empieza a programar en Java. Cuando se avanza en conocimientos, se pueden crear más tipos de ventanas.



Vamos a comentar la analogía entre lo que hemos hecho usando BlueJ y lo que hicimos en epígrafes anteriores usando el bloc de notas y Java directamente.

PASO DADO	ANTES	AHORA
Escritura de código fuente	Con el bloc de notas	Con el editor de un IDE (BlueJ)
Compilado	Ventana consola DOS del sistema operativo mediante la instrucción <i>javac Ejemplo.java</i>	Pulsando el botón u opción de menú “Compile” del IDE.
Ejecución	Ventana consola DOS del sistema operativo mediante la instrucción <i>java Ejemplo</i>	Invocando la ejecución con el IDE mediante la opción <i>void main (String [] arg)</i>
Resultado	Se muestra Hola Java en la ventana consola DOS del sistema operativo.	Se muestra Hola Java en la ventana consola del IDE.

Comparación entre dos formas de ejecutar un programa en Java

Esta comparación nos sirve para entender que independientemente del IDE (BlueJ, Eclipse, NetBeans, etc.) que usemos, **los procesos que “en el fondo” tienen lugar son los mismos**. La gestión se podrá hacer en un tipo de ventana u otra, y pulsando unos botones u otros, pero lo que realmente permite que estos entornos den lugar a resultados es el sistema Java (destacando compilador y máquina virtual) que se encuentra en nuestro ordenador.

Próxima entrega: CU00618B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

COMENTARIOS EN EL LENGUAJE DE PROGRAMACIÓN JAVA. CONCEPTO DE BLOQUE DE CÓDIGO.

Cuando escribimos código en general es útil realizar comentarios explicativos. Los comentarios no tienen efecto como instrucciones para el ordenador, simplemente sirven para que cuando un programador lea el código pueda comprender mejor lo que lee.



Más adelante estudiaremos que Java tiene un sistema normalizado de comentarios (javadoc), ahora simplemente vamos a indicar cómo introducir dos tipos de comentarios:

- **Comentario multilínea:** se abre con el símbolo /* y se cierra con el símbolo */
- **Comentario en una línea o al final de una línea:** se introduce con el símbolo //

Prueba a introducir comentarios en el código. Aquí te mostramos ejemplos de cómo hacerlo:

```
/*
 * Este es el primer programa en un IDE del curso Java aprenderaprogramar.com
 * Creado el 29/03/2027
 */

// A continuación el código del programa
public class Ejemplo {
    public static void main(String[] arg) {
        System.out.println("Hola Java"); //Usamos esta sintaxis para mostrar mensajes por pantalla
    }
}
```

Otro concepto que vamos a introducir ahora es el de bloque. Vamos a denominar bloque al código comprendido dentro de los símbolos { y }. Por lo tanto en el programa anterior podríamos distinguir dos bloques. En ocasiones indicaremos junto al símbolo de cierre del bloque algún comentario que nos permita saber a qué bloque cierra ese símbolo. Esto no es necesario ni obligado, pero cuando los programas son largos nos puede ayudar. Ejemplo:

```
public class Ejemplo {
    public static void main(String [] arg) {
        System.out.println("Hola Java");
    } //Cierre del main
} //Cierre de la clase
```

Próxima entrega: CU00619B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONCEPTO DE OBJETOS Y CLASES EN JAVA. DEFINICIÓN DE INSTANCIA. EJEMPLOS

Hemos realizado una primera incursión en Java pero para proseguir se nos hace indispensable hablar de conceptos fundamentales de la programación orientada a objetos: **objetos** y **clases**. Estos términos parecen resultarnos familiares.

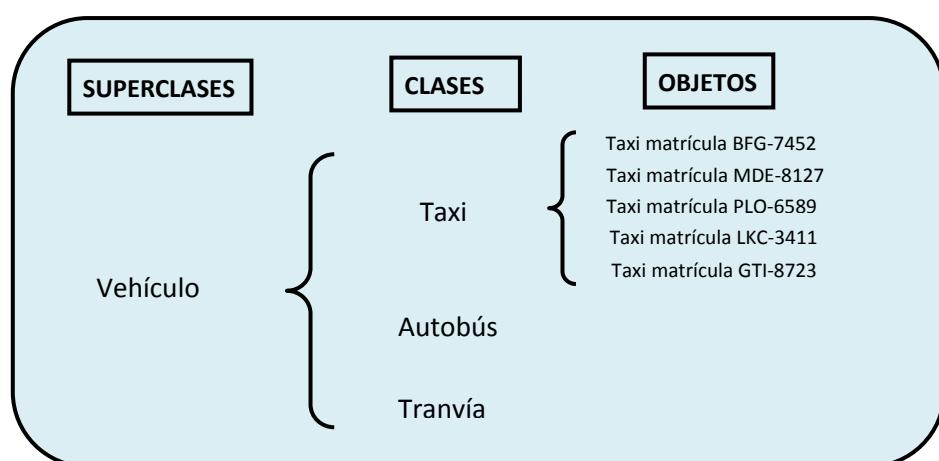


En la vida diaria podemos pensar en objetos como una manzana o un libro y podemos distinguir clases de cosas: por ejemplo clases de plantas. Sin embargo, en programación el término objeto y el término clase no guardan una correlación exacta con el significado de estas palabras en la vida diaria. Podemos buscar ciertas similitudes e incluso hacer analogías didácticas. Pero no trates de buscar siempre equivalencias entre objetos y clases en programación con objetos y clases de la vida diaria porque esa correspondencia exacta no existe y te llevará a confusión. Al escribir un programa en un lenguaje orientado a objetos tratamos de modelar un problema del mundo real pensando en objetos que forman parte del problema y que se relacionan entre sí. Daremos ahora una primera definición de objeto y clase, que tendremos que ir matizando conforme avancemos en el curso.

Objeto: entidad existente en la memoria del ordenador que tiene unas propiedades (atributos o datos sobre sí mismo almacenados por el objeto) y unas operaciones disponibles específicas (métodos).

Clase: abstracción que define un tipo de objeto especificando qué propiedades (atributos) y operaciones disponibles va a tener.

Estas definiciones son quizás poco clarificadoras. Con un ejemplo vamos a entenderlo mejor. En primer lugar pensemos en un programa que trata de gestionar datos sobre los vehículos de transporte público de una ciudad, por ejemplo México D.F.



En este ejemplo hemos considerado que el problema consta de tres tipos de vehículo: taxi, autobús y tranvía, y que esos tipos los denominamos clases. ¿Qué haríamos en Java para definir una clase? Indicar sus propiedades y operaciones (métodos) disponibles, por ejemplo:

```

Clase Taxi {
    Propiedades:
        Matrícula identificativa
        Distrito en el que opera
        Tipo de motor diesel o gasolina
        Coordenadas en las que se ubica

    Operaciones disponibles:
        Asignar una matrícula
        Asignar un distrito
        Asignar un tipo de motor
        Ubicar en unas coordenadas
}

```

El haber definido así el taxi significará que todo objeto de tipo Taxi que creemos tendrá una matrícula identificativa, un distrito en el que opera, un tipo de motor y unas coordenadas en las que se ubica. La creación de un objeto sería algo así como: “*Crear un objeto Taxi con matrícula BFG-7452, distrito Norte, tipo de motor Diesel y coordenadas Desconocidas.*”

El uso de una operación sobre un objeto sería algo así como: “*Taxi BFG-7452 → Ubicar en coordenadas (X = 128223, Y = 877533)*”. Las operaciones en Java se denominan métodos, veremos cómo se definen más adelante.

Decimos que un objeto es una instancia de una clase. Por ejemplo el taxi matrícula BFG-7452 es una instancia de la clase Taxi. Varios objetos (p.ej. taxis) de una misma clase decimos que constituyen instancias múltiples de la clase. Más adelante veremos que tanto una clase como un objeto en Java pueden representar otras cosas además de lo que ahora hemos explicado, pero todo a su tiempo.

EJERCICIO

Considera que queremos representar mediante un programa Java los aviones que operan en un aeropuerto. Crea un esquema análogo al que hemos visto para vehículos, pero en este caso para aviones. Define cuáles podrían ser las clases y cuáles podrían ser algunos objetos de una clase. Para comprobar la corrección de tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00620B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

VISUALIZAR CLASES Y CREAR OBJETOS JAVA CON BLUEJ. INVOCAR MÉTODOS.

Vamos a crear varias clases y objetos en nuestro entorno de desarrollo. Para ello repetiremos el proceso que seguimos con la clase Ejemplo pero escribiendo el siguiente código:



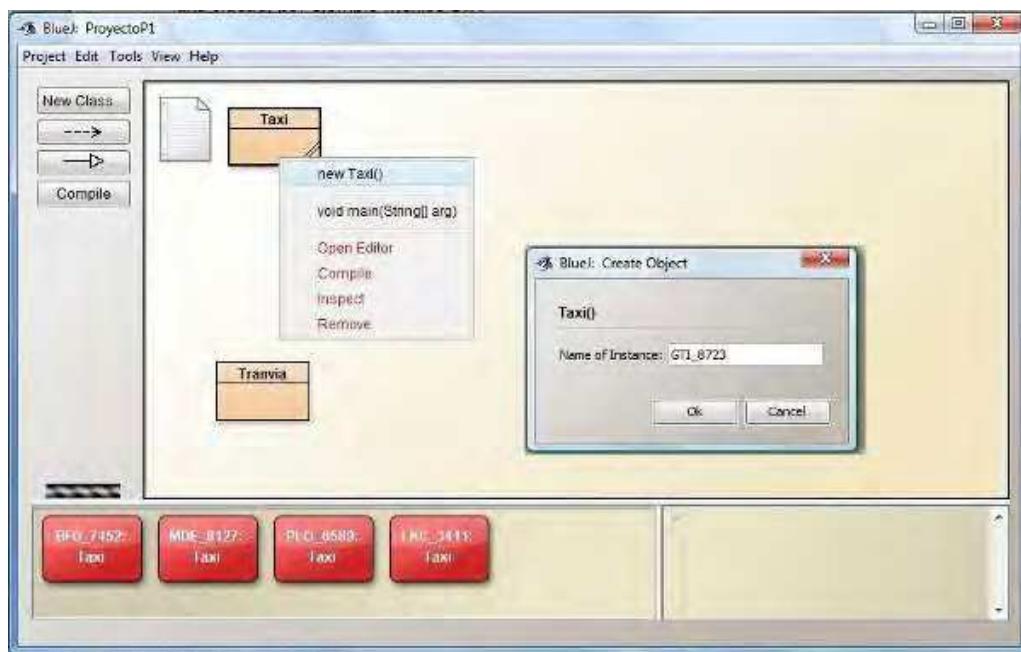
```
//Ejemplo aprenderaprogramar.com
public class Taxi {
    public static void main (String[ ] arg) {
        System.out.println ("Soy un taxi");
    } //Cierre del main
} //Cierre de la clase
```

En otra clase escribiremos:

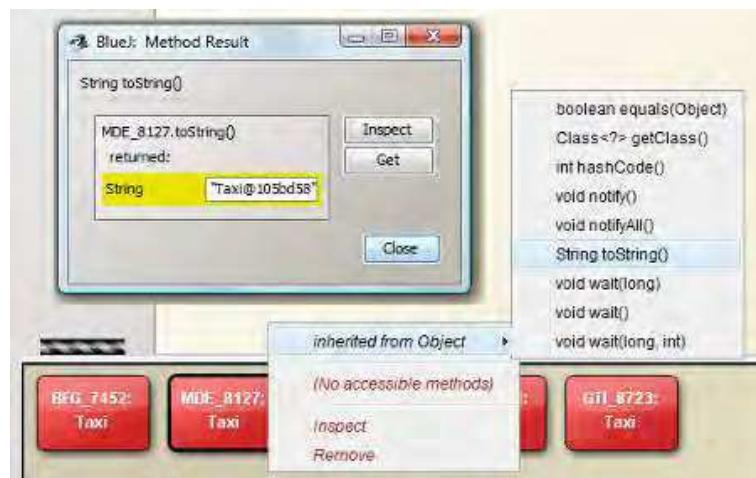
```
//Ejemplo aprenderaprogramar.com
public class Tranvia {
    public static void main (String[ ] arg) {
        System.out.println ("Soy un tranvía");
    } //Cierre del main
} //Cierre de la clase
```

La clase Ejemplo que teníamos en nuestra ventana de BlueJ la eliminaremos pulsando sobre ella y con botón derecho elegimos “Remove” (también podemos hacerlo a través del menú Edit → Remove). Ahora tenemos dos clases y dos iconos de clase: Taxi y Tranvia. Para crear objetos taxi pinchamos sobre el icono Taxi y con botón derecho elegimos new Taxi(). Nos aparece una ventana que nos pide el nombre del objeto y escribimos BFG_7452 (usamos guión bajo porque no se admite guión medio). Repetimos varias veces el proceso y vamos creando distintos objetos taxi. Cada vez que creamos un taxi nos aparece en la parte inferior izquierda un rectángulo rojo con un texto como BFG_7452: Taxi. Este rectángulo representa un objeto taxi. El espacio en la parte inferior izquierda de la pantalla donde se van mostrando los objetos creados de esta manera se denomina “Banco de objetos” (Object Bench).

Hemos creado varios objetos taxi. Fíjate que **cuando solo tenemos definida la clase no existen objetos**: los objetos hay que crearlos para que existan.



¿Qué pueden hacer nuestros objetos taxis? Pues prácticamente nada, porque todavía no hemos escrito código que nos permita hacer algo. Vamos simplemente a pedir a cada taxi que nos diga el espacio de memoria que ocupa: para ello pulsamos sobre el icono de un objeto taxi (por ejemplo MDE_8127) y con botón derecho seleccionamos *Inherited from Object → String toString()*. Se nos mostrará algo parecido a lo que mostramos en la siguiente imagen:



Nos aparece: *returned "Taxi@105bd58"* (no tiene por qué coincidir con estos dígitos). Prueba a hacer lo mismo con otros objetos Taxi. Verás que cada objeto devuelve una cadena *Taxi@.....* distinta. Esto significa que cada objeto ocupa un espacio de memoria distinto y a ese espacio de memoria se le denomina de esa manera un poco extraña. Hacer esto ha sido posible porque al crear objetos en Java por defecto se dispone de algunos métodos comunes a cualquier objeto. Nosotros en realidad todavía no hemos definido métodos propios para los objetos tipo Taxi.

En un programa, cada objeto de tipo Taxi podría informar de su posición, de su tipo de motor, etc. o ser llamado para que tome determinada posición (coordenadas), o para modificar el tipo de motor que tiene establecido, entre muchas otras cosas.

Prueba a crear objetos tranvía y a consultar el identificador de su espacio de memoria. Prueba también a borrar objetos del banco de objetos. Para ello pulsa sobre su ícono y con el botón derecho del ratón elige la opción “Remove”.

Por último, vamos a indicar un convenio que sigue la mayoría de los programadores: **a las diferentes clases les pondremos nombres que comiencen por mayúscula** como “Taxi”. Por el contrario, a los objetos les pondremos nombres que comiencen por minúscula. En nuestro caso podríamos haber nombrado a los objetos como taxi_BFG_7452 para respetar el convenio. Seguir esta norma no es obligada, pero hacerlo es recomendable para mantener un buen estilo de programación. Las empresas suelen mantener un manual de estilo que sirve de guía para que todos los programadores que trabajen en un proyecto sigan unas normas comunes.

Próxima entrega: CU00621B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

TIPOS DE DATOS (VARIABLES) EN JAVA. TIPOS PRIMITIVOS (INT, ETC.) Y OBJETO.

Los primeros lenguajes de programación no usaban objetos, solo variables. Una variable podríamos decir que es **un espacio de la memoria del ordenador a la que asignamos un contenido** que puede ser un valor numérico (sólo números, con su valor de cálculo) o de tipo carácter o cadena de caracteres (valor alfanumérico que constará sólo de texto o de texto mezclado con números).



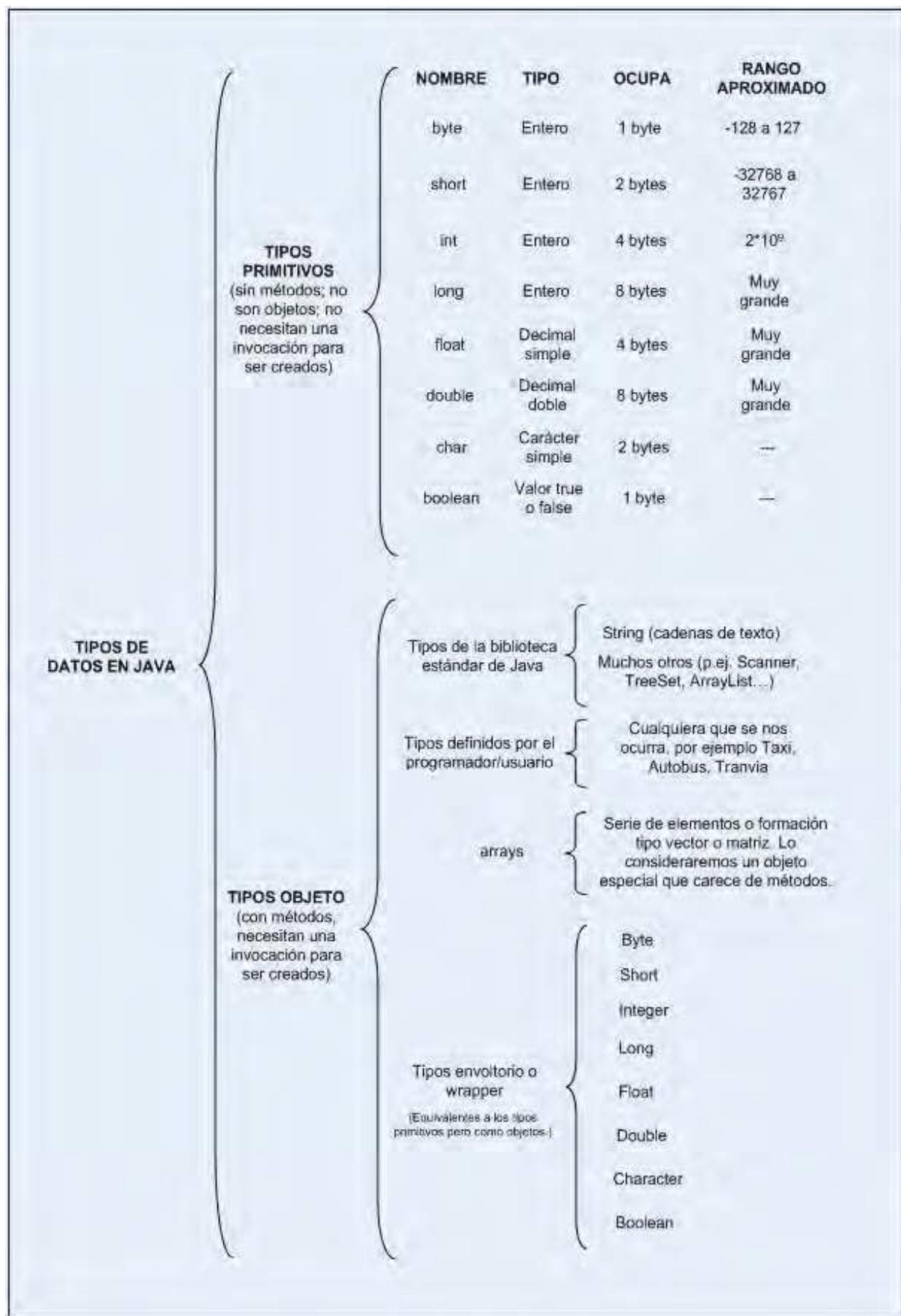
Como ejemplo podemos definir una variable a que contenga 32 y esto lo escribimos como `a = 32`. Posteriormente podemos cambiar el valor de a y hacer `a = 78`. O hacer "a" equivalente al valor de otra variable "b" así: `a = b`.

Dado que antes hemos dicho que un objeto también ocupa un espacio de memoria: **¿en qué se parecen y en qué se diferencia un objeto de una variable?** Consideraremos que las variables son entidades elementales: un número, un carácter, un valor verdadero o falso... mientras que los objetos son entidades complejas que pueden estar formadas por la agrupación de muchas variables y métodos. Pero ambas cosas ocupan lo mismo: un espacio de memoria (que puede ser más o menos grande).

En los programas en Java puede ser necesario tanto el uso de datos elementales como de datos complejos. Por eso en Java se usa el término "Tipos de datos" para englobar a cualquier cosa que ocupa un espacio de memoria y que puede ir tomando distintos valores o características durante la ejecución del programa. Es decir, en vez de hablar de tipos de variables o de tipos de objetos, hablaremos simplemente de tipos de datos. Sin embargo, a veces "coloquialmente" no se utiliza la terminología de forma estricta: puedes encontrarte textos o páginas web donde se habla de una variable en alusión a un objeto.

En Java diferenciamos dos tipos de datos: por un lado, los tipos primitivos, que se corresponden con los tipos de variables en lenguajes como C y que son los datos elementales que hemos citado. Por otro lado, los tipos objeto (que normalmente incluyen métodos).

Veamos los tipos de datos en Java sobre un esquema de síntesis:



Esquema de síntesis de tipos de datos en Java

Este esquema no es necesario aprendérselo de memoria en todos sus detalles, aunque sí lo iremos memorizando poco a poco a medida que lo utilicemos, por lo menos hasta tener en nuestra cabeza los nombres de todos los tipos primitivos y envoltorio y sus características (si son objetos o no y su rango aproximado). Vamos a comentar distintas cuestiones:

1. Un objeto es una cosa distinta a un tipo primitivo, aunque “porten” la misma información. Tener siempre presente que los objetos en Java tienen un tipo de tratamiento y los tipos primitivos, otro. Que en un momento dado contengan la misma información no significa en ningún caso que sean lo mismo. Iremos viendo las diferencias entre ambos poco a poco. De momento, recuerda que el tipo primitivo es algo elemental y el objeto algo complejo. Supón una cesta de manzanas en la calle: algo elemental. Supón una cesta de manzanas dentro de una nave espacial (considerando el conjunto nave + cesta): algo complejo. La información que portan puede ser la misma, pero no son lo mismo.

2. ¿Para qué tener esa aparente duplicidad entre tipos primitivos y tipos envoltorio? Esto es una cuestión que ataña a la concepción del lenguaje de programación. Tener en cuenta una cosa: un tipo primitivo es un dato elemental y carece de métodos, mientras que un objeto es una entidad compleja y dispone de métodos. Por otro lado, de acuerdo con la especificación de Java, es posible que necesitemos utilizar dentro de un programa un objeto que “porte” como contenido un número entero. Desde el momento en que sea necesario un objeto habremos de pensar en un envoltorio, por ejemplo Integer. Inicialmente nos puede costar un poco distinguir cuándo usar un tipo primitivo y cuándo un envoltorio en situaciones en las que ambos sean válidos. Seguiremos esta regla: usaremos por norma general tipos primitivos. Cuando para la estructura de datos o el proceso a realizar sea necesario un objeto, usaremos un envoltorio.

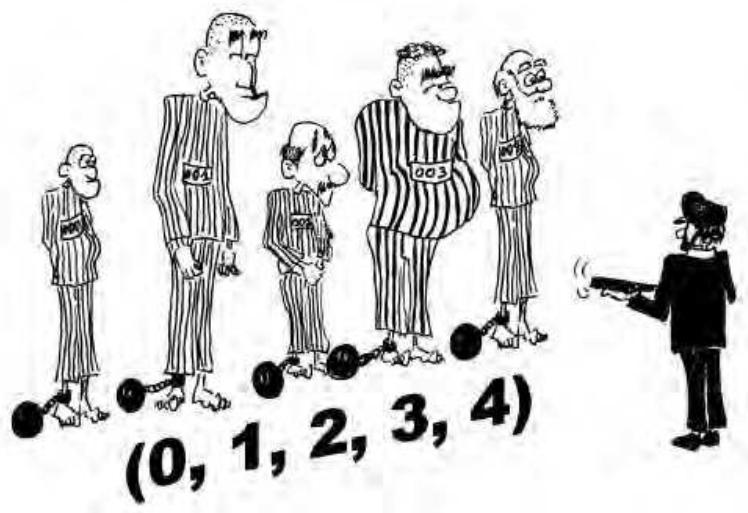
3. Los nombres de tipos primitivos y envoltorio se parecen mucho. En realidad, excepto entre int e Integer y char y Character, la diferencia se limita a que en un caso la inicial es minúscula (por ejemplo double) y en el otro es mayúscula (Double). Esa similitud puede confundirnos inicialmente, pero hemos de tener muy claro qué es cada tipo y cuándo utilizar cada tipo.

4. Una cadena de caracteres es un objeto. El tipo String en Java nos permite crear objetos que contienen texto (palabras, frases, etc.). El texto debe ir siempre entre comillas. Muchas veces se cree erróneamente que el tipo String es un tipo primitivo por analogía con otros lenguajes donde String funciona como una variable elemental. En Java no es así.

5. Hay distintos tipos primitivos enteros. ¿Cuál usar? Por norma general usaremos el tipo int. Para casos en los que el entero pueda ser muy grande usaremos el tipo long. Los tipos byte y short los usaremos cuando tengamos un mayor dominio del lenguaje.

6. ¿Cuántos tipos de la biblioteca estándar de Java hay? Cientos o miles. Es imposible conocerlos todos.

7. ¿Un array es un objeto? Los arrays los consideraremos objetos especiales, los únicos objetos en Java que carecen de métodos.



Concepto de array: serie de elementos, cada uno de los cuales lleva asociado un índice numérico 0, 1, 2, 3, ... , n-1

Próxima entrega: CU00622B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EJEMPLOS DE USO DE TIPOS DE DATOS (VARIABLES) EN JAVA. DECLARACIÓN, INICIALIZACIÓN.

Vamos a ver ejemplos de uso de tipos de datos en Java. Para ello nos valdremos primeramente de algunos tipos primitivos usados habitualmente como son int (entero), String (cadena de caracteres), boolean (valor booleano verdadero o falso), float (decimal simple), etc.



Aquí mostramos ejemplos de uso de tipos de datos en Java:

```
public class Ejemplo2 {
    private int precio; // Las instrucciones y declaraciones finalizan con ;
    private int importe_acumulado;
    private String profesor;
    private String aula;
    private int capacidad;
    private boolean funciona;
    private boolean esVisible;
    private float diametro;
    private float peso;
    private short edad;
    private long masa;
    private char letra1;
} //Cierre de la clase
```

Hemos declarado variables de tipo primitivo u objeto usando la sintaxis *private tipoElegido nombreVariable;*

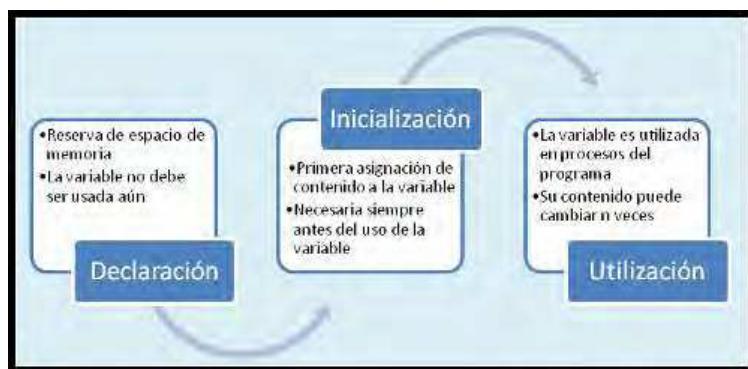
La palabra clave *private* es un indicador de en qué ámbito del programa va a estar disponible la variable. Supón que el programa es un edificio con gente trabajando y que hay elementos en el edificio, por ejemplo una impresora, que pueden tener un uso: individual para una persona, colectivo para un grupo de personas, colectivo para todas las personas de una planta, colectivo para todas las personas de un departamento aunque estén en varias plantas, o colectivo para todo el edificio. Pues bien, las variables en Java van a quedar disponibles para su uso en ciertas partes del programa según especifiquemos con las palabras clave *public*, *private*, *protected*, *package*, etc. Lo veremos más adelante, ahora simplemente nos interesa ver cómo declarar variables y usaremos de forma preferente la palabra clave *private*.

El hecho de **declarar una variable implica que se reserva un espacio de memoria para ella**, pero no que ese espacio de memoria esté ocupado aunque pueda tener un contenido por defecto. Ten en cuenta que en Java no puedes aplicar algunas normas que rigen en otros lenguajes, como que al

declarar una variable entera ésta contendrá por defecto el valor cero. En Java esta situación puede dar lugar a errores de compilación: una variable entera no debemos suponer que contenga nada. Para que contenga algo debemos asignarle un contenido. Veamos ejemplos de asignación de contenido:

```
Precio = 42; // Entero tipo int. Un número sin punto decimal se interpreta normalmente como int.
importe_acumulado = 210; // Entero tipo int
profesor = "Ernesto Juárez Pérez"; // Tipo String
aula = "A-44"; // Tipo String
capacidad = 1500; // Entero tipo int
funciona = true; // Tipo boolean
esVisible = false; // Tipo boolean
diametro = 34.25f; // Tipo float. Una f o F final indica que es float.
peso = 88.77; // Tipo double. Un número con punto decimal se interpreta normalmente como double.
edad = 19; // Entero tipo short
masa = 178823411L; // Entero tipo long. Una l o L final indica que es long.
letra1 = 'h'; // Tipo char (carácter). Se escribe entre comillas simples.
```

Hemos planteado la declaración de variables en primer lugar y la asignación de contenido en segundo lugar y por separado porque será una forma habitual de trabajar en Java.



Esto no significa que en determinadas ocasiones no podamos declarar e inicializar (asignar contenido) simultáneamente. Por ejemplo: `int edad = 19;` será una expresión válida y que utilizaremos en determinadas ocasiones, según iremos viendo.

La inicialización es un paso importante de cara a permitir un uso seguro de una variable. Es tan importante, que en general plantearemos que se haga como paso previo a cualquier otra cosa. Por ejemplo, si pensamos utilizar una variable denominada `precio` lo primero que haremos será establecer un valor de `precio` o, si no lo conocemos o lo vamos a establecer más adelante, estableceremos explícitamente un valor por defecto: por ejemplo `precio = - 99;` ó `precio = 0;`. Utilizar una variable sin haberla inicializado es una práctica no recomendada en Java (mal estilo de programación) que puede dar lugar a errores o al malfuncionamiento de los programas.

Próxima entrega: CU00623B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ ES UNA CLASE JAVA? ATRIBUTOS, CONSTRUCTOR Y MÉTODOS

Hasta ahora hemos visto pequeños fragmentos de código de ejemplo. Vamos a tratar de escribir un código más ajustado a la realidad de la programación Java. Para ello vamos a definir de qué partes consta normalmente una clase Java. Las partes habituales las identificamos en este esquema:



Clae Taxi { --- > EL NOMBRE DE LA CLASE

Propiedades: --- > También denominadas atributos o campos (fields)

Matrícula identificativa

Distrito en el que opera

Tipo de motor diesel o gasolina

Constructor de la clase --- > Definición de qué ocurre cuando se crea un objeto del tipo definido por la clase

Operaciones disponibles: --- > Métodos de la clase

Asignar una matrícula

Asignar un distrito

Asignar un tipo de motor

}

Esto vamos a transformarlo en código usando un ejemplo. Para ello abre un nuevo proyecto en BlueJ y crea en él una clase denominada Taxi. Escribe en ella este código, aunque no entiendas algunas partes de él.

```
//Esta clase representa un taxi. --> Comentario general que puede incluir: cometido, autor, versión, etc...
public class Taxi { //El nombre de la clase

    private String ciudad; //Ciudad de cada objeto taxi
    private String matricula; //Matrícula de cada objeto taxi
    private String distrito; //Distrito asignado a cada objeto taxi
    private int tipoMotor; //tipo de motor asignado a cada objeto taxi. 0=desconocido, 1 = gasolina, 2 = diesel

    //Constructor: cuando se cree un objeto taxi se ejecutará el código que incluyamos en el constructor
    public Taxi () {
        ciudad = "México D.F.";
        matricula = "";
        distrito = "Desconocido";
        tipoMotor = 0;
    } //Cierre del constructor ...el código continúa ...
}
```

```

//Método para establecer la matrícula de un taxi
public void setMatricula (String valorMatricula) {
    matricula = valorMatricula; //La matrícula del objeto taxi adopta el valor que contenga valorMatricula
} //Cierre del método

//Método para establecer el distrito de un taxi
public void setDistrito (String valorDistrito) {
    distrito = "Distrito " + valorDistrito; //El distrito del objeto taxi adopta el valor indicado
} //Cierre del método

public void setTipoMotor (int valorTipoMotor) {
    tipoMotor = valorTipoMotor; //El tipoMotor del objeto taxi adopta el valor que contenga valorTipoMotor
} //Cierre del método

//Método para obtener la matrícula del objeto taxi
public String getMatricula () { return matricula; } //Cierre del método

//Método para obtener el distrito del objeto taxi
public String getDistrito () { return distrito; } //Cierre del método

//Método para obtener el tipo de motor del objeto taxi
public int getTipoMotor () { return tipoMotor; } //Cierre del método

} //Cierre de la clase

```

Pulsa el botón Compile y comprueba que no haya ningún error.

Repasemos lo que hemos hecho: hemos creado una clase denominada *Taxi*. El espacio comprendido entre la apertura de la clase y su cierre, es decir, el espacio entre los símbolos { y } de la clase, se denomina cuerpo de la clase.

Hemos dicho que todo objeto de tipo *Taxi* tendrá los mismos atributos: una matrícula (cadena de caracteres), un distrito (cadena de caracteres) y un tipo de motor (valor entero 0, 1 o 2 representando *desconocido*, *gasolina* o *diesel*). Los atributos los definiremos normalmente después de la apertura de la clase, fuera de los constructores o métodos que puedan existir.

Hemos definido que cualquier objeto *Taxi* que se cree tendrá, inicialmente, estos atributos: como matrícula una cadena vacía; como distrito “Desconocido”; y como tipo de motor 0, que es el equivalente numérico de *desconocido*. La sintaxis que hemos utilizado para el constructor es *public nombreDeLaClase { ... }*

Por otro lado, hemos establecido que todo objeto *Taxi* podrá realizar estas operaciones: recibir un valor de matrícula y quedar con esa matrícula asignada (*setMatricula*); recibir un valor de distrito y quedar con ese distrito asignado (*setDistrito*); recibir un valor de tipo de motor y quedar con ese valor asignado (*setTipoMotor*). Devolver su matrícula cuando se le pida (*getMatricula*); devolver su distrito cuando se le pida (*getDistrito*); devolver su tipo de motor cuando se le pida (*getTipoMotor*).

Para crear objetos Taxi pinchamos sobre el icono Taxi de la clase y con botón derecho elegimos new Taxi(). Nos aparece una ventana que nos pide el nombre del objeto. Crea 5 objetos Taxi denominados taxi1, taxi2, taxi3, taxi4 y taxi5. Cada objeto Taxi tiene tres atributos: matricula, distrito y tipoMotor. En total tendremos 5 taxis x 3 atributos = 15 atributos.

Hemos dicho que un objeto es una instancia de una clase: por eso a los atributos que hemos definido se les denomina “variables de instancia”, porque cada instancia es “portadora” de esos atributos. También es frecuente utilizar el término “campos de la clase” como equivalente. Cada clase tendrá sus campos específicos. Por ejemplo, si una clase representa una moneda sus campos pueden ser *país, nombreMoneda, valor, diámetro, grosor*. Si una clase representa una persona sus campos pueden ser *nombre, apellidos, dni, peso y altura*.

¿Para qué nos sirve la clase? Para crear objetos de tipo Taxi. Por eso se dice que en Java una clase define un tipo. Recordamos ahora la definición de clase que habíamos dado previamente: “**Clase**: abstracción que define un tipo de objeto especificando qué propiedades y operaciones disponibles va a tener.”

¿Por qué la clase, el constructor y los métodos se declaran public y los atributos private? Esto lo discutiremos más adelante. De momento, nos basta con saber que declararemos las clases, constructores y métodos precedidos de la palabra clave *public*, y que esta palabra afecta a en qué partes del programa o por parte de quién se va a poder acceder a ellos (igual que en el edificio con personas trabajando decíamos que una impresora podía tener un uso restringido a el personal de un departamento).

¿El orden campos → constructor → métodos es obligatorio? No, pero a la hora de programar hemos de ser metódicos y evitar el desorden. Muchos programadores utilizamos este orden a la hora de escribir clases, así que no está mal acostumbrarnos a seguir este orden.

¿Por qué en unos casos un método ocupa una línea y en otros varias líneas? Simple cuestión de espacio. Puedes escribirlo como quieras, siempre que quede bien presentado y legible. Hemos de tener claro que un método consta de dos partes: un encabezado o línea inicial y un cuerpo o contenido dentro de las llaves { }. En este curso muchas veces escribiremos métodos en una sola línea, o varias instrucciones en una sola línea, para ahorrar espacio. Sin embargo, en el trabajo como programadores el ahorro de espacio es poco relevante frente a la claridad. Lo importante es que el código sea claro.

¿Por qué establecemos el tipo de motor con un entero en vez de con un texto tipo String? A veces podemos definir las variables de diferentes maneras. En este caso nos resultaría también válido usar un String en vez de un int. Pero ten en cuenta una cosa: a los ordenadores les resulta más fácil analizar y manejar números que palabras. Si tenemos cien taxis en realidad no va a resultar demasiado importante elegir un texto o un número. Pero si tenemos cien mil sí puede ser relevante elegir un tipo numérico porque va a acelerar el procesado de datos.

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo Persona. Define una clase Persona análoga a la que hemos visto para taxis, pero en este caso considerando los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), numeroDocumentoidentidad (String). Define un constructor y los métodos para poder establecer y obtener los valores de los atributos. Compila el código para comprobar que no presenta errores. Para comprobar la corrección de tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00624B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

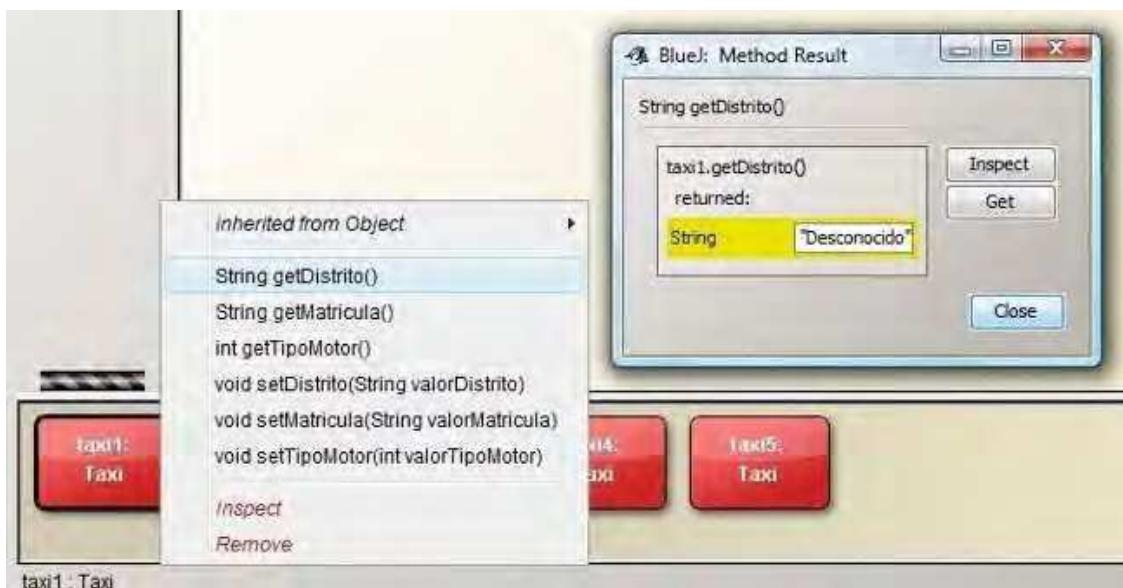
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

MÉTODOS EN JAVA TIPO PROCEDIMIENTO (...VOID) Y TIPO FUNCIÓN (...RETURN)

Volvamos sobre los objetos taxi creados y que deben aparecer en el banco de objetos en la parte inferior izquierda de la pantalla del entorno de desarrollo. Si pulsamos con botón derecho del ratón sobre ellos se nos despliegan los métodos (operaciones) disponibles para cada objeto.



Pulsa sobre el taxi1 y elige la opción *String getDistrito*.



La ejecución de un método se denomina habitualmente **“invocación” del método o “llamada” al método**. Los métodos disponibles los define la clase, pero se invocan sobre cada objeto en particular. Al invocar el método *getDistrito()* se nos abre una ventana de BlueJ denominada *Method Result* donde nos indica: *returned String “Desconocido”*. Le hemos pedido al objeto que nos diga cuál es su distrito y nos devuelve “Desconocido”. La razón para ello es que en el constructor de la clase incluimos una línea de inicialización de distrito para todo objeto de tipo Taxi con el valor “Desconocido”. Si en vez de ese valor hubiésemos establecido otro, ese sería el que ahora obtendríamos.

Cierra esa ventana y repite el proceso eligiendo ahora la opción *void setDistrito (String valorDistrito)*. En el recuadro donde solicita el distrito escribe “Oeste”. No olvides incluir las comillas obligadas por tratarse de un String. Seguidamente, vuelve a invocar el método *getDistrito* y comprueba el resultado obtenido.



Los métodos que hemos definido en la clase Taxi podemos clasificarlos de la siguiente manera:

- a) **Métodos tipo función:** son métodos que nos devuelven algo. Un método es tipo función si comienza con un tipo (no consideramos ahora la palabra clave public). Por ejemplo *String getDistrito()* comienza con el tipo String lo que significa que nos devuelve una cadena de caracteres, mientras que *int getTipoMotor()* comienza con el tipo int lo que significa que nos devuelve un entero. Tener en cuenta que un método tipo función devuelve solo un dato u objeto como resultado, no varios. La devolución del resultado se expresa con la palabra clave *return* seguida del dato u objeto a devolver, por ejemplo *return tipoMotor;*. La sentencia *return* implica que termina la ejecución del código en el método y estará típicamente en la línea final. De existir una línea por detrás de una sentencia *return*, nunca llegaría a ejecutarse (tendremos que matizar esto porque el uso de condicionales nos permitirá tener más de un *return* en un método). De forma genérica:

```
//Comentario descriptivo de qué hace el método
public tipoValorDevuelto nombreDelMétodo (tipo parámetro1, tipo parámetro2...) {

    Código del método
    return ResultadoQueDevuelveElMétodo;
}
```

- b) **Métodos tipo procedimiento:** son métodos que realizan ciertas operaciones sin devolver un valor u objeto concreto. Un método es tipo procedimiento si comienza con la palabra clave *void* (que traducido del inglés viene siendo “vacío” o “nulo”). En estos casos podemos decir que el tipo de retorno es *void*. De forma genérica:

```
//Comentario descriptivo de qué hace el método
public void nombreDelMétodo (tipo parámetro1, tipo parámetro2...) {

    Código del método
}
```

En general un método con tipo de retorno *void* no llevará sentencia *return*, aunque en Java se permite que un método de este tipo incluya la sentencia *return;*. Si ocurre esto, la sentencia da lugar a que el método finalice en ese punto sin ejecutar ninguna línea más de código. Solo tiene sentido su uso asociado a que se cumplan ciertas condiciones.

¿Puede un método ser al mismo tiempo función y procedimiento? Formalmente debemos evitarlo. Podríamos tratar de agrupar operaciones, pero es una situación que trataremos de evitar. Nos plantearemos como objetivo que todo método tipo función se centre en devolvernos aquello que nos interesa sin realizar otros procesos. Igualmente buscaremos que todo procedimiento realice un proceso concreto y no varios. Cada método debe realizar una tarea concreta, específica y bien definida. Un método no debe ocuparse de dos tareas.

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo DiscoMusical. Define una clase DiscoMusical análoga a la que hemos visto para taxis, pero en este caso considerando los siguientes atributos de clase: titulo (String), autor (String), añoEdicion (int), formato (String), digital (boolean). Define un constructor y los métodos para poder establecer y obtener los valores de los atributos. Compila el código para comprobar que no presenta errores. Crea un objeto y comprueba sus métodos como hemos hecho con los objetos Taxi. Para comprobar la corrección de tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00625B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

MÉTODOS EN JAVA CON Y SIN PARÁMETROS

Al igual que hicimos con distrito, continúa invocando los métodos del objeto taxi1 para establecer sus valores de matrícula a “BFG-7452” y tipo de motor a 2. Los métodos que hemos definido en la clase Taxi podemos clasificarlos de otra manera:



- a) **Métodos que solicitan parámetros:** son métodos que nos piden algo (uno o varios datos u objetos). Es decir, el método para ejecutarse necesita que se le envíe un parámetro de un tipo concreto. Los métodos que solicitan parámetros se identifican porque en los paréntesis finales incluyen uno o varios términos, por ejemplo (*String valorMatricula*) nos indica que el método requiere un parámetro de tipo *String*. Fíjate que en este caso el parámetro es un objeto tipo *String*. En cambio la expresión (*int valorTipoMotor*) nos indica que el parámetro es un tipo primitivo *int*. Un método podría requerir varios parámetros para lo cual se indican separados por comas. Por ejemplo *public int costeVivienda (int superficie2, String zonaCiudad, int calidadesMedias)*.
- b) **Métodos sin parámetros:** son métodos que no piden ningún dato u objeto para ejecutarse. Un método sin parámetros se identifica porque sus paréntesis finales están vacíos. Estos métodos no necesitan recibir información para ejecutarse.

Una cuestión importante en los métodos con parámetros es el tipo requerido. Prueba a introducir un dato erróneo. Por ejemplo, introduce el texto “gasolina” cuando te pida el tipo de motor de un taxi. El resultado es que se produce un error porque el tipo recibido no coincide con el tipo esperado. El correcto manejo de tipos es un aspecto importante en la mayor parte de lenguajes de programación, incluido Java, y será algo a lo que debamos prestar especial atención siempre que estemos desarrollando código.

Los constructores funcionan en buena medida de forma similar a los métodos y también podemos definir constructores con parámetros y constructores sin parámetros. El sentido de que un constructor pida uno o varios parámetros es tener en cuenta esos parámetros a la hora de crear un objeto. Abordaremos los constructores con más detenimiento un poco más adelante.

Próxima entrega: CU00626B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

MÉTODOS CONSULTORES O DE ACCESO (GETTERS) Y MÉTODOS MODIFICADORES (SETTERS) EN JAVA

Continuamos con el uso de métodos en Java. Prueba a establecer distintos valores de matrícula, distrito y tipo de motor para los diferentes objetos Taxi. Prueba también a, una vez establecidos unos valores para un objeto, volver a cambiarlos por otros valores distintos.



Hay aún otra manera de clasificar los métodos que hemos definido para la clase Taxi:

- a) **Métodos modificadores:** llamamos métodos modificadores a aquellos métodos que dan lugar a un cambio en el valor de uno o varios de los atributos del objeto.
- b) **Métodos consultores u observadores:** son métodos que devuelven información sobre el contenido de los atributos del objeto sin modificar los valores de estos atributos.

Cuando se crea una clase es frecuente que lo primero que se haga sea establecer métodos para consultar (de ahí su denominación de consultores) sus atributos y estos métodos suelen ir precedidos del prefijo get (getMatricula, getDistrito, etc.) por lo que muchas veces se alude coloquialmente a ellos como “métodos get” o “getters”.

Se suele proceder de igual forma con métodos que permitan establecer los valores de los atributos. Estos métodos suelen ir precedidos del prefijo set (setMatricula, setDistrito, etc.) por lo que muchas veces se alude coloquialmente a ellos como “métodos set” o “setters”. Los métodos set son un tipo de métodos modificadores, porque cambian el valor de los atributos de un objeto.

¿Puede un método ser al mismo tiempo modificador y consultor? Es posible, pero es una situación que trataremos de evitar. Nos plantearemos como objetivo que cada método haga una cosa específica y no varias al mismo tiempo.

Próxima entrega: CU00627B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

ESTADO DE UN OBJETO DURANTE LA EJECUCIÓN DE UN PROGRAMA JAVA

Hemos comprobado que un objeto tiene unos atributos. Además, si tenemos métodos disponibles para ello, podemos cambiar varias veces el valor de estos atributos. Por ejemplo, al objeto taxi1 le podemos establecer inicialmente distrito “Oeste”, luego distrito “Norte”, “Sur”, etc.



Llamamos estado de un objeto al conjunto de valores de los atributos del objeto en un momento dado. Sobre el objeto taxi1, pulsa botón derecho y elige la opción *Inspect*.



El resultado es que se nos abre una ventana que se denomina “Inspector de objetos” y que refleja su estado. En este caso, nos indica que el valor de sus atributos es “BFG-7452” para la matrícula del taxi, “Distrito Oeste” para el distrito y 2 para el tipo de motor. Prueba a modificar el estado del objeto taxi1 utilizando alguno de sus métodos sin cerrar la ventana de inspección. Comprobarás que los datos se actualizan automáticamente a medida que vamos realizando cambios. Prueba a inspeccionar el estado de los distintos objetos Taxi que hayas creado.

Próxima entrega: CU00628B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PARÁMETROS FORMALES Y PARÁMETROS ACTUALES EN MÉTODOS Y CONSTRUCTORES JAVA

Hemos visto que un método (o constructor) puede requerir un parámetro, con un tipo y nombre concretos. Por ejemplo tipo *String* y nombre del parámetro *valorDistrito*. A su vez *valorDistrito* podría ir tomando distintos valores a lo largo de una ejecución del método, por ejemplo “Norte”, “Oeste”, “Sur”, etc.



Vamos a definir dos conceptos relacionados con los parámetros:

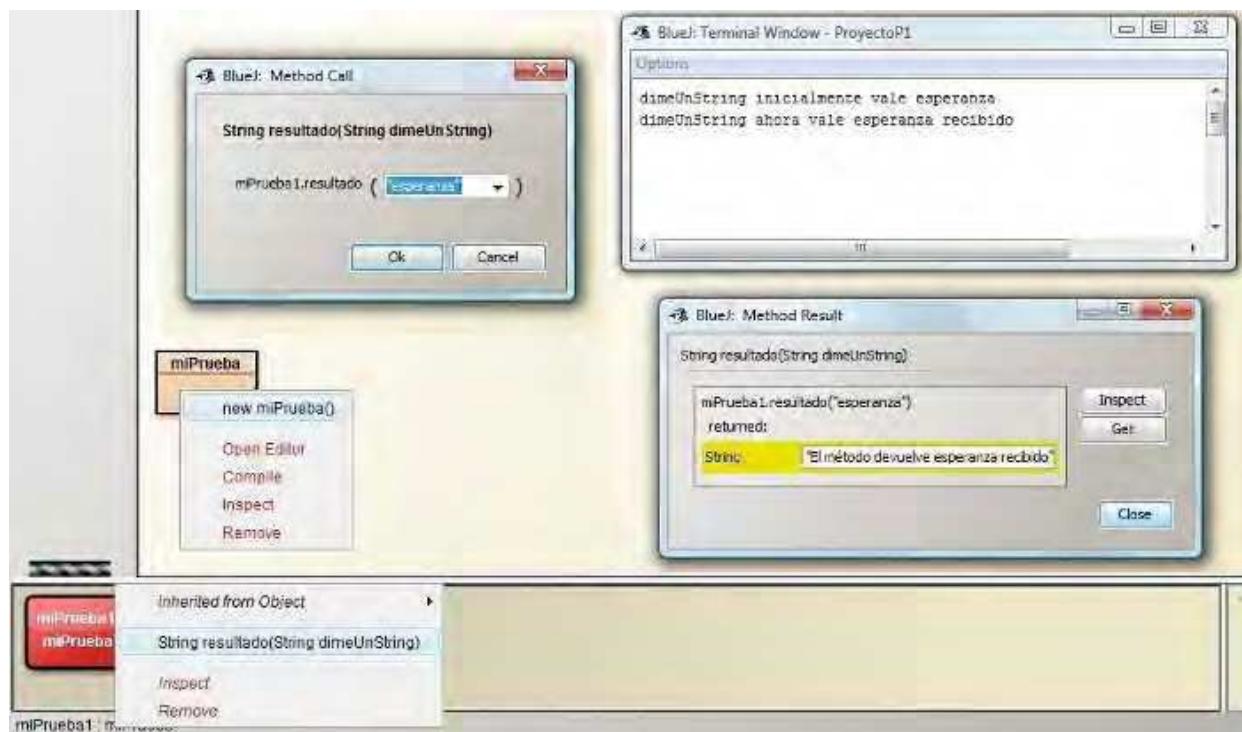
- a) **Parámetro formal:** es el nombre de un parámetro de un constructor o método tal y como se define en su cabecera, por ejemplo *valorDistrito* es un parámetro formal.
- b) **Parámetro actual:** es el valor concreto que tiene un parámetro en un momento dado.

Prueba a escribir y compilar el siguiente código:

```
//Esta clase es una prueba. Curso aprenderaprogramar.com Java desde cero
public class miPrueba {

    public String resultado (String dimeUnString) {
        System.out.println ("dimeUnString inicialmente vale " + dimeUnString);
        dimeUnString = dimeUnString + " recibido";
        System.out.println ("dimeUnString ahora vale " + dimeUnString);
        return "El método devuelve " + dimeUnString;
    }
}
```

Ahora crea un objeto de tipo *miPrueba* y ejecuta el método introduciendo una cadena, por ejemplo “esperanza”. El resultado será algo similar a esto:



El parámetro formal `dimeUnString` es de tipo `String` y inicialmente toma el valor que le hayamos pasado al constructor cuando creamos el objeto. Luego su valor actual pasa a ser el valor pasado al constructor + “recibido”. Finalmente el método nos devuelve una cadena de la que forma parte el parámetro.

Conclusión interesante: los parámetros de métodos o constructores también almacenan valores que pueden ir cambiando. Por eso diremos que también son variables al igual que los campos, aunque conceptualmente sean cosas distintas.

Ten en cuenta una cosa: un campo es una variable que está de forma permanente asociada a un objeto y cuyo ámbito (lugares donde podemos usarla) es toda la clase. Su tiempo de vida es indefinido mientras exista el objeto al que está ligado. En cambio **un parámetro formal tiene un ámbito limitado al método o constructor y tiempo de vida limitado** al tiempo durante el cual se ejecuta el método o constructor. En este sentido, decimos que un parámetro es un tipo de variable local (solo podemos usarla dentro del método) y temporal (se crea al comenzar la ejecución del método y se destruye al terminar su ejecución).

Para evitar confusiones, en general trataremos de evitar modificar el contenido de un parámetro dentro del código de un método.

Próxima entrega: CU00629B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

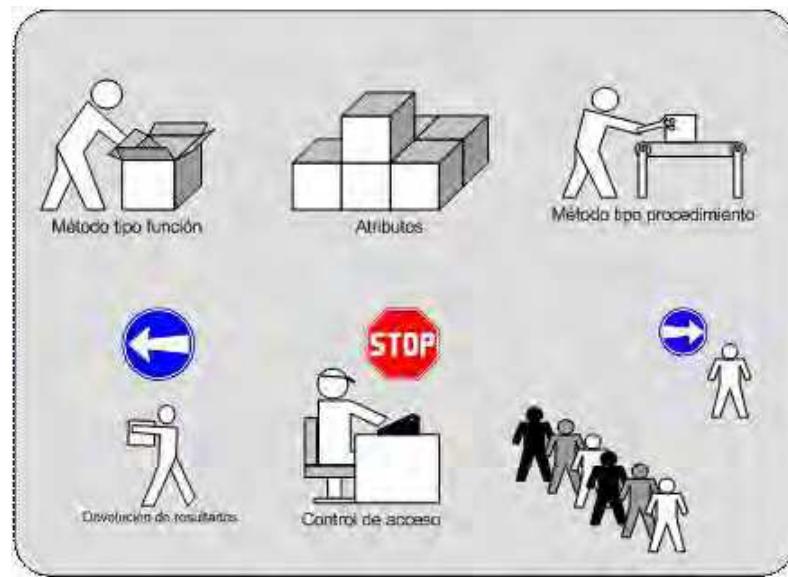
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

COMPRENDER EL CONCEPTO Y FILOSOFÍA DE MÉTODOS Y CLASES EN JAVA

Hemos visto la forma habitual de organizar una clase en Java. Trataremos ahora de reflexionar sobre por qué se hacen así las cosas. Vamos a explicarlo sobre un esquema gráfico: lee el siguiente texto al mismo tiempo que miras el esquema “Funcionamiento de métodos y clases en Java” incluido a continuación.



A un objeto le llegan muchas solicitudes, que hemos representado como personas en una cola frente a un control de acceso. El control de acceso representa que todo lo que se pretenda hacer está sometido a supervisión: cualquiera no puede dar órdenes y quien puede dar órdenes no puede hacer lo que quiera. Por ejemplo, puede haber una solicitud para establecer que la ciudad del objeto pase a ser “Mérida”. El control de acceso verifica si es posible tal acción, y al no ser posible ya que no se ha definido ningún método que permita modificar el atributo *ciudad*, la petición es rechazada. Puede haber otra solicitud que pida información al objeto sobre qué valor tiene su atributo *ciudad*. El control de acceso comprueba que no existe un método que permita tal acción y la petición es rechazada.



Esquema del funcionamiento de métodos y clases en Java

Puede haber una solicitud que pida información al objeto sobre su atributo *distrito*: es una operación permitida y se entrega la información (el estado del objeto no cambia).

Dentro del objeto el trabajo es ordenado. Hay métodos específicamente encargados de preparar información para servirla (tipo función). Otros métodos se encargan de hacer manipulaciones en el objeto.

En resumen, **la clase define qué se puede hacer y qué no se puede hacer**. El objeto funciona como una entidad que trabaja de forma ordenada y especializada. Este planteamiento difiere del que existía en los lenguajes de programación primigenios, donde se podían hacer modificaciones a variables desde cualquier lugar y sin control. Esto en sí no es un problema, el problema venía a posteriori cuando los programadores hacían un mal uso de esa “libertad”.

En la programación actual, se trabaja en equipos en algunos casos de cientos o miles de personas que desarrollan código por separado. Para evitar que se hagan cosas inadecuadas, se utilizan mecanismos de control que favorecen que la programación sea de calidad. En Java no existe un “control de acceso” tal y como lo hemos representado en nuestro esquema, pero sí existen distintos mecanismos de control como la declaración del ámbito y accesibilidad a las variables y clases (*public, private, protected*, etc.) y la obligación de uso de métodos para realizar operaciones sobre objetos impidiendo su manipulación directa.

Próxima entrega: CU00630B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

SIGNATURA DE UN MÉTODO. INTERFAZ O INTERFACE.

El esquema planteado en relación a la filosofía de clases y métodos en Java tiene otras implicaciones. Una de ellas es que la persona que llega con una solicitud u orden no puede más que hacer una entrega de esa solicitud u orden, pero no puede entrar al objeto a realizar manipulaciones o coger cosas.



Lo que pasa dentro del objeto no se ve. Este principio, denominado “**ocultamiento de la información**”, se manifiesta de distintas maneras y es muy relevante en programación orientada a objetos. Veamos una aplicación de este principio en relación a lo que hemos explicado hasta ahora. Consideremos lo siguiente:

```
float calcularCapacidadDeposito ()
```

Podemos interpretar que esto es el encabezado de un método. Y podemos extraer cierta información relevante: el método devuelve un valor numérico tipo decimal simple, el método sirve para calcular la capacidad de un depósito, y el método no requiere parámetros. El encabezado de un método se denomina **signatura del método** e informa de varias cosas:

- a) Si el método es **tipo función o tipo procedimiento**.
- b) El **tipo del valor devuelto**, si es un método tipo función.
- c) El **nombre del método**.
- d) Los **parámetros requeridos** (ninguno, uno o varios) y sus tipos.

Veamos otro ejemplo de signatura:

```
float calcularCapacidadDeposito (float valorDiametro, float valorAltura)
```

Con esta información sabemos lo que hace el método, pero no cómo lo hace. En general a este tipo de información que nos dice qué se hace pero no cómo, lo denominamos interfaz o interface. Esta palabra tiene distintos significados en Java que iremos viendo poco a poco. De momento, nos quedaremos con la idea de que la signatura es una interfaz de un método, porque informa de lo que hace, pero no nos dice cómo lo hace (queda oculta su implementación o desarrollo). En muchas ocasiones trabajaremos conociendo sólo la signatura de métodos y desconociendo su código de desarrollo. Esto no será

problema: mientras los procesos funcionen y estén bien programados, no nos va a hacer falta conocer todo el código.

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo DiscoMusical. Define las signaturas para métodos dentro dicha clase e indica si deberán ser procedimientos o funciones para los siguientes objetivos planteados:

- 1) Obtener la duración de una canción expresada como un número decimal en minutos (por ejemplo podría ser 3,22 minutos) recibiendo como dato el número de canción dentro del disco.
- 2) Pedir a un administrador de una compañía discográfica que introduzca la duración de todas las canciones (dichas duraciones se almacenarán como información del objeto). Es decir, el método se encargará de saludar al usuario y pedirle que vaya introduciendo una por una la duración de las canciones.
- 3) Obtener la duración del disco completo (suma de las duraciones de cada una de las canciones).
- 4) Añadir una canción al disco recibiendo como información para ello el título de la canción y su duración (se añadirá como última canción en el disco. Si el disco tenía por ejemplo 10 canciones, al añadirse una canción pasará a tener 11).

Para comprobar si tu solución es correcta puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00631B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

GUARDAR LOS PROYECTOS JAVA. COPIAS DE SEGURIDAD Y GESTIÓN DE VERSIONES.

Muchos entornos de desarrollo, incluido BlueJ, trabajan guardando continuamente los cambios realizados en los proyectos. Si estamos trabajando y pulsamos directamente sobre el aspa de cierre del editor o sobre el aspa de cierre del programa, los cambios quedan guardados automáticamente.



Esto tiene la ventaja de que no tenemos que preocuparnos de estar pulsando un botón “guardar” periódicamente como puede ser habitual en editores de texto como Microsoft Word. Pero también tiene un inconveniente, que es el que modificaciones introducidas nos generen un perjuicio que no podamos evitar al no saber localizar exactamente el código “malo” que genera distorsiones en nuestro programa.

Cuando estamos trabajando en el desarrollo de un programa extenso, que pueda requerir días, semanas o meses de dedicación, conviene tomar precauciones para no perder código desarrollado por bloqueos del ordenador, fallos en soportes como discos duros o introducción de código “malo”.

En primer lugar y como precaución, básica, realiza una copia de seguridad de los archivos del proyecto a un soporte externo (como pendrive, disco duro o servidor externo) diariamente si el proyecto es importante. **Cada cierto tiempo, realiza una copia de seguridad** en un cd o dvd y manténla en un lugar seguro. En segundo lugar, un proyecto evoluciona desde cero líneas de código hasta miles de líneas de código. Entre el principio y el final, hay muchos estados del proyecto. Es conveniente manejar estos estados como si se tratara de versiones o prototipos de programa de forma que en cualquier momento podamos recuperar un estado anterior si así lo deseamos. Existen herramientas de software libre para el manejo de versiones como *Subversion* o *Git*. Pueden ser de utilidad, pero no vamos a entrar aquí a explicar su uso. Vamos a proponer un manejo bastante más sencillo y rústico que consiste en lo siguiente:

1. Cuando comiences a trabajar en un proyecto importante crea primero una carpeta para almacenar las versiones o prototipos con el nombre general del proyecto (por ejemplo, “Cobra”). Luego comienza a trabajar con el entorno de desarrollo creando un proyecto dentro de la carpeta creada anteriormente y ponle al proyecto un nombre terminado en _01. Por ejemplo *Cobra_01*.
2. Cuando termines de trabajar en el proyecto en ese día cierra el entorno de desarrollo y busca la carpeta que contiene los archivos correspondientes al proyecto. Cópiala y renómbrala para que el nombre termine en _011, por ejemplo *Cobra_011*. Al día siguiente abre y usa como proyecto de trabajo el de numeración más grande, de modo que la carpeta anterior permanezca como copia de seguridad. BlueJ te permitirá seguir trabajando sin problema.
3. En días sucesivos vete repitiendo el proceso renombrando con terminaciones en _012, _013, etc.

4. Por otro lado, en un archivo de texto (que puede ser creado con el bloc de notas) al que puedes denominar changelog.txt, vete registrando las características de cada versión y los cambios relevantes introducidos respecto a la anterior. Por ejemplo una anotación podría ser la siguiente:

En 0_29 está implementado el constructor de la clase TruckSystem y probado que la generación del sistema de truckers es correcta. También está implementado el método getTruckBySize (String size) y probado que se localiza correctamente el truck solicitado.

Nombre	Tamaño	Tipo
Cobra_01		Carpeta de archivos
Cobra_011		Carpeta de archivos
Cobra_012		Carpeta de archivos
Cobra_013		Carpeta de archivos
Cobra_014		Carpeta de archivos
Cobra_015		Carpeta de archivos
Cobra_016		Carpeta de archivos
changelog.txt	5 KB	Documento de texto

Realiza copias de seguridad periódicas de todo el sistema de archivos. De este modo, ante un fallo en el ordenador o soporte, no perderás el trabajo realizado. También te permitirá volver atrás si no estás satisfecho con la evolución del código y retomar el proyecto desde un punto dado. Por último, cuando tengas un programa que creas terminado, crea la copia de los archivos renombrando el proyecto con la terminación 1_0 y para sucesivas versiones utiliza terminaciones como 1_01, 1_02, 1_03, etc.

El sistema que hemos descrito es bastante casero frente a herramientas para mantenimiento profesional de versiones, pero nos puede servir perfectamente para dar nuestros primeros pasos como programadores Java evitando sustos indeseados.

Como último apunte, haremos referencia a la estabilidad de un entorno de desarrollo. Un IDE es, al fin y al cabo, software, es decir, un programa o aplicación. Al igual que cualquier programa se puede “bloquear”, “colgar” o dar algún tipo de problemas. BlueJ es un entorno bastante estable, pero ocasionalmente puede fallar por bloqueos, envío de mensajes de error de compilación sin motivo, visualizaciones extrañas o cierres inesperados. En primer lugar, indicarte que en el 99 % de los casos el problema será tuyo: no trates de achacártelo al entorno. En los casos excepcionales en que sospeches que se pueda tratar de un fallo del programa, procede en primer lugar a cerrarlo y abrirlo de nuevo para ver si el problema desaparece. En segundo lugar a apagar y reiniciar el ordenador. Si se te presenta un problema persistente, consulta en el sitio oficial del IDE o en foros de programación.

Próxima entrega: CU00632B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

IMPRIMIR POR CONSOLA EN JAVA (SYSTEM.OUT). CONCATENAR CADENAS. NOTACIÓN DE PUNTO.

En Java hay algunos objetos que existen por defecto (en cualquier entorno de desarrollo, llámese Eclipse, NetBeans, BlueJ, etc.). Uno de ellos es el objeto denominado *System.out*. Este objeto dispone de un método llamado *println* que nos permite imprimir algo por pantalla en una ventana de consola.



La sintaxis básica es: *System.out.println ("Mensaje a mostrar");*

Ten en cuenta que la primera S de *System.out* es mayúscula. Si la escribes minúscula obtendrás un error de compilación. Tenlo presente porque cualquier pequeño error de escritura (el simple cambio de una letra) en el nombre de un objeto, variable, método, etc. puede dar lugar a errores. Además, BlueJ en ocasiones te señalará el lugar del error pero en otras ocasiones no lo hará con exactitud y tendrás que buscarlo con paciencia. Ten también presente que los espacios dentro de las comillas cuentan, es decir, el resultado de escribir ("Mensaje a mostrar"); no es el mismo que el de escribir ("Mensaje a mostrar");.

Si queremos incluir variables concatenamos usando el símbolo + de esta manera:

System.out.println ("El precio es de " + precio + " euros");

El símbolo + se usa para concatenar cadenas de texto, o variables que representen texto. ¿Qué ocurre si introducimos en una concatenación un número o una variable que no sea una cadena? Java por defecto realizará la conversión de aquello que hayamos concatenado a texto. Por ejemplo:

System.out.println ("El precio es de " + 42 + " euros");

Se verá en pantalla como: *El precio es de 42 euros*. Fíjate que hemos incluido los espacios necesarios antes y después del número para evitar que por pantalla nos apareciera *El precio es de42euros*.

Si queremos imprimir una línea en blanco escribiremos esto: *System.out.println ();*

El mismo resultado se obtiene escribiendo *System.out.println ("");*

Escribir esto es válido: *System.out.println ("Mi nombre es " + "Juan");* ya que al fin y al cabo estamos concatenando dos cadenas, aunque sea más lógico escribir ("Mi nombre es Juan");

Fíjate que la sintaxis que estamos usando responde al siguiente esquema:

nombreDelObjeto.nombreDelMétodo (parámetro1, parámetro 2, ...);

En el caso de un objeto Taxi, una invocación de un método podría ser `taxi1.getDistrito()`. Esta invocación devuelve una cadena, por lo que podríamos escribir:

```
System.out.println ("El distrito del taxi 1 es " + taxi1.getDistrito() );
```

Esta forma de invocar los métodos se denomina “notación de punto” porque se basa en escribir el nombre del objeto seguido de un punto y el nombre del método con los parámetros que procedan. La notación de punto es un aspecto básico de Java y otros lenguajes de programación.

Otro uso sería el siguiente: `taxi1.distrito = "Norte"`. En este caso no estamos invocando a un método, sino a un atributo. ¿Cómo sabemos si se trata de una invocación a un método o a un atributo? Simplemente hemos de fijarnos en si la invocación termina con unos paréntesis o no. Si no hay paréntesis, se trata de un atributo. Si los hay, se trata de un método.

Una última cuestión a comentar sobre la concatenación de cadenas es su uso en el resultado que puede devolver un método tipo función. Supongamos que `precio` es una variable de tipo `int` con valor 42. Si escribimos como línea final de un método `return precio;` estamos devolviendo 42, un entero. Si escribimos `return "precio"` estamos devolviendo la cadena de texto “precio” y no la variable `precio`. ¿Qué ocurre si escribimos una sentencia como `return "" + precio;`? **Unas comillas que se abren y cierran son una cadena**, aunque sean una cadena vacía. El operador `+` después de una cadena, concatena como cadena aquello que venga a continuación, ya que Java realiza por defecto la conversión a cadena. Por tanto en este caso el valor devuelto sería “42” como cadena de texto y no como valor numérico. Fíjate que esto es una forma de convertir un tipo numérico a tipo texto.

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo `Medico` (que representa a un médico de un hospital). Define una clase `Medico` considerando los siguientes atributos de clase: `nombre` (`String`), `apellidos` (`String`), `edad` (`int`), `casado` (`boolean`), `numeroDocumentoIdentidad` (`String`), `especialidad` (`String`). Define un constructor asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. En cada método, incluye una instrucción para que se muestre por consola un mensaje informando del cambio. Por ejemplo si cambia la especialidad del médico, debe aparecer un mensaje que diga: “Ha cambiado la especialidad del médico de nombre La nueva especialidad es: ...”. Compila el código para comprobar que no presenta errores, crea un objeto, usa sus métodos y comprueba que aparezcan correctamente los mensajes por consola. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00633B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

OPERADORES ARITMÉTICOS EN JAVA. EL OPERADOR % (MOD) O RESTO DE DIVISIÓN.

En Java disponemos de los operadores aritméticos habituales en lenguajes de programación como son suma, resta, multiplicación, división y operador que devuelve el resto de una división entre enteros (en otros lenguajes denominado operador mod o módulo de una división):



OPERADOR	DESCRIPCIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de una división entre enteros (en otros lenguajes denominado mod)

Operadores aritméticos en Java

Destacar que el operador % es de uso exclusivo entre enteros. $7 \% 3$ devuelve 1 ya que el resto de dividir 7 entre 3 es 1. Al valor obtenido lo denominamos módulo (en otros lenguajes en vez del símbolo % se usa la palabra clave *mod*) y a este operador a veces se le denomina “operador módulo”.

Aunque en otros lenguajes existe un operador de exponentiación, en Java no es así. Para calcular una potencia podemos hacer varias cosas:

- Recurrir a multiplicar n veces el término. Por ejemplo \min^3 lo podemos calcular como $\min * \min * \min$. Obviamente esto no es práctico para potencias de exponentes grandes.
- Usar un bucle que dé lugar a la repetición de la operación multiplicación n veces, o usar un método que ejecute la operación. Estas opciones las comentaremos más adelante.
- Usar herramientas propias del lenguaje que permiten realizar esta operación. Esta opción la comentaremos más adelante.

Las operaciones con operadores siguen un **orden de prelación o de precedencia** que determinan el orden con el que se ejecutan. Si existen expresiones con varios operadores del mismo nivel, la operación se ejecuta de izquierda a derecha. Para evitar resultados no deseados, en casos donde pueda existir duda se recomienda el uso de paréntesis para dejar claro con qué orden deben ejecutarse las operaciones. Por ejemplo, si dudas si la expresión $3 * a / 7 + 2$ se ejecutaría en el orden que tú deseas, especifica el orden deseado utilizando paréntesis: por ejemplo $3 * ((a / 7) + 2)$.

EJERCICIO

Define una clase Medico considerando los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), numeroDocumentoidentidad (String), especialidad (String). Define un constructor asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. Define un método de nombre "calculoParaMultiploEdad" que no recibe parámetros y es tipo procedimiento cuyo cometido será el siguiente: determinar cuántos años faltan para que la edad del médico sea múltiplo de 5 y mostrar un mensaje informativo por pantalla. Por ejemplo si el médico tiene 22 años deberá en primer lugar obtener el resto de la división de 22 entre 5, que es 2. Ahora obtendrá los años que faltan para que el médico tenga una edad múltiplo de 5, que serán $5 - 2 = 3$ años. A continuación deberá mostrar un mensaje por consola del tipo: "El médico de nombre ... con especialidad ... tendrá una edad múltiplo de 5 dentro de ... años". Compila el código para comprobar que no presenta errores, crea un objeto, usa sus métodos y comprueba que aparezcan correctamente los mensajes por consola, y que cuando cambias la edad aparece correctamente el tiempo que falta para que la edad sea múltiplo de 5. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00634B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

OPERADORES LÓGICOS PRINCIPALES EN JAVA

En Java disponemos de los operadores lógicos habituales en lenguajes de programación como son “es igual”, “es distinto”, menor, menor o igual, mayor, mayor o igual, and (y), or (o) y not (no). La sintaxis se basa en símbolos como veremos a continuación y cabe destacar que hay que prestar atención a no confundir == con = porque implican distintas cosas.



OPERADOR	DESCRIPCIÓN
<code>==</code>	Es igual
<code>!=</code>	Es distinto
<code><, <=, >, >=</code>	Menor, menor o igual, mayor, mayor o igual
<code>&&</code>	Operador and (y)
<code> </code>	Operador or (o)
<code>!</code>	Operador not (no)

Operadores lógicos principales en Java

El operador || se obtiene en la mayoría de los teclados pulsando ALT GR + 1, es decir, la tecla ALT GR y el número 1 simultáneamente.

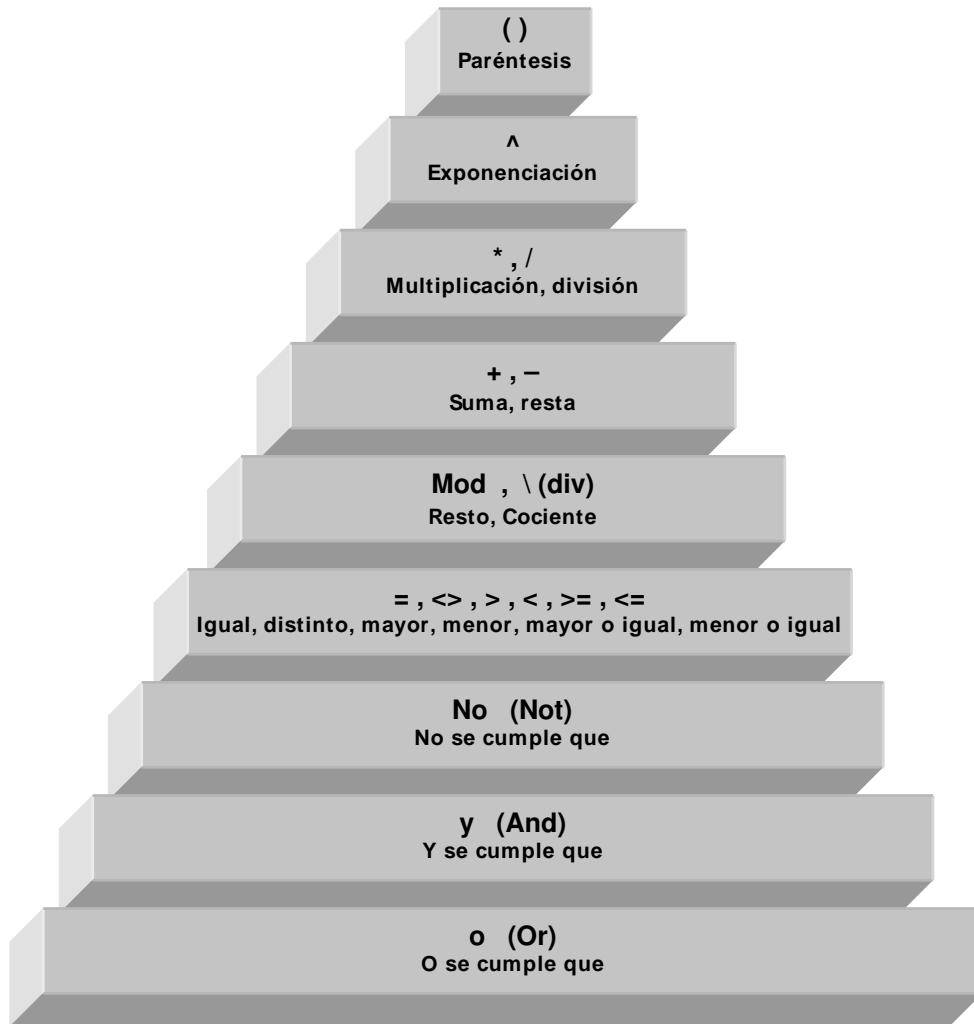
Los operadores && y || se llaman **operadores en cortocircuito** porque si no se cumple la condición de un término no se evalúa el resto de la operación. Por ejemplo: (a == b && c != d && h >= k) tiene tres evaluaciones: la primera comprueba si la variable a es igual a b. Si no se cumple esta condición, el resultado de la expresión es falso y no se evalúan las otras dos condiciones posteriores.

En un caso como (a < b || c != d || h <= k) se evalúa si a es menor que b. Si se cumple esta condición el resultado de la expresión es verdadero y no se evalúan las otras dos condiciones posteriores.

El operador ! recomendamos no usarlo hasta que se tenga una cierta destreza en programación. Una expresión como (!esVisible) devuelve false si (esVisible == true), o true si (esVisible == false). En general existen expresiones equivalentes que permiten evitar el uso de este operador cuando se desea.

ORDEN DE PRIORIDAD, PRELACIÓN O PRECEDENCIA

Los operadores lógicos y matemáticos tienen un orden de prioridad o precedencia. Este es un esquema general que indica el orden en que deben evaluarse en la mayoría de los lenguajes de programación:



Una expresión como $A+B == 8 \&\& A-B == 1$ siendo $A = 3$ y $B = 5$ supondrá que se evalúa primero $A+B$ que vale 8, luego se evalúa $A-B$ que vale -2. Luego se evalúa si se cumple que la primera operación es cierta y luego si la segunda también es cierta, resultando que no, por lo que la expresión es falsa.

EJERCICIO

Dadas las variables de tipo int con valores A = 5, B = 3, C = -12 indicar si la evaluación de estas expresiones daría como resultado verdadero o falso:

- a) $A > 3$
- b) $A > C$
- c) $A < C$
- d) $B < C$
- e) $B \neq C$
- f) $A == 3$
- g) $A * B == 15$
- h) $A * B == -30$
- i) $C / B < A$
- j) $C / B == -10$
- k) $C / B == -4$
- l) $A + B + C == 5$
- m) $(A+B == 8) \&& (A-B == 2)$
- n) $(A+B == 8) || (A-B == 6)$
- o) $A > 3 \&& B > 3 \&& C < 3$
- p) $A > 3 \&& B \geq 3 \&& C < -3$

Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00635B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

SENTENCIA DE ASIGNACIÓN EN JAVA. OPERADOR DE ASIGNACIÓN COMPUESTA.

Vamos a ver cómo realizar asignación de contenido a variables. De momento hablaremos solo de variables, ya que el tratamiento de objetos conviene hacerlo con cautela. Además veremos el operador de asignación compuesta, un operador que es de uso opcional.



La asignación de contenido a variables se hace en Java de la siguiente manera: `variable = expresión;`

Por ejemplo `precio = precioBase + 0.35;` En toda asignación debe haber coincidencia de tipos. Si `peso` es una variable de tipo `int` y `sobrePeso` es una variable de tipo `double` ó `boolean` debemos evitar sentencias como `sobrePeso = 2 * peso;` Aunque hay ocasiones en que la falta de coincidencia de tipos será asumida por Java, debemos tratar de evitarlo en todo caso porque introduce inseguridad en la programación.

En Java una sentencia como `saldo = saldo + cantidad;` se puede escribir también así: `saldo += cantidad;`

`+=` se denomina operador de asignación compuesta y sirve para sumarle una cantidad al valor de una variable. También se admite el uso del operador `-=` que en vez de sumar lo que hace es restar. El uso de los operadores de asignación compuesta es opcional: hay programadores que los usan y otros que no.

Un ejemplo de asignación compuesta con `-=` sería: `saldo -= precioCaja * cantidadCajas;`

Próxima entrega: CU00636B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

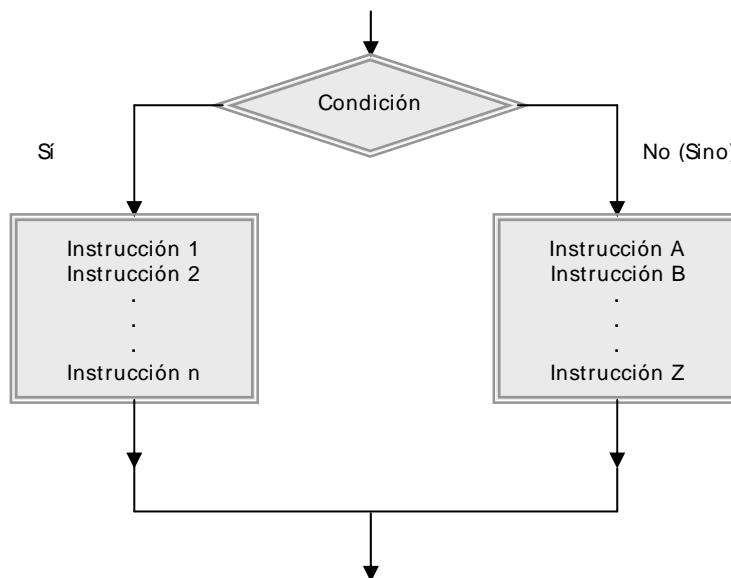
ESTRUCTURA O ESQUEMA DE DECISIÓN EN JAVA. IF ELSE , IF ELSE IF.

La instrucción *if ... else* permite controlar qué procesos tienen lugar, típicamente en función del valor de una o varias variables, de un valor de cálculo o booleano, o de las decisiones del usuario. La sintaxis a emplear es:



```
if (condición) {
    instrucciones
} else {
    instrucciones
}
```

Esquemáticamente en forma de diagrama de flujo:



La cláusula *else* (no obligatoria) sirve para indicar instrucciones a realizar en caso de no cumplirse la condición. **Java admite escribir un else y dejarlo vacío:** *else { }*. El *else* vacío se interpreta como que contemplamos el caso pero no hacemos nada en respuesta a él. Un *else* vacío no tiene ningún efecto y en principio carece de utilidad, no obstante a veces es usado para remarcar que no se ejecuta ninguna acción cuando se alcanza esa situación.

Cuando se quieren evaluar distintas condiciones una detrás de otra, se usa la expresión *else if { }*. En este caso no se admite *elseif* todo junto como en otros lenguajes. De este modo, la evaluación que se produce es: si se cumple la primera condición, se ejecutan ciertas instrucciones; si no se cumple,

comprobamos la segunda, tercera, cuarta... n condición. Si no se cumple ninguna de las condiciones, se ejecuta el else final en caso de existir.

```
//if sencillo
if ( admitido == true) { System.out.println ("Se ha admitido el valor"); }

//if else sencillo
if ( admitido == true) {
    System.out.println ("Se ha admitido el valor");
} else {
    System.out.println ("No se ha admitido el valor");
}

//if con else if y cláusula final else
if (DesplazamientoX == 0 && DesplazamientoY == 1) {
    System.out.println ("Se procede a bajar el personaje 1 posición");
}
else if (DesplazamientoX == 1 && DesplazamientoY == 0) {
    System.out.println ("Se procede a mover el personaje 1 posición a la derecha"); }

else if (DesplazamientoX == -1 && DesplazamientoY == 0) {
    System.out.println ("Se procede a mover el personaje 1 posición a la izquierda");
}
else {
    System.out.println ("Los valores no son válidos");
}
```

Intenta compilar este código en una clase. Para ello declara la clase, declara las variables que intervienen, inicialízalas en un constructor, e incorpora tres métodos que se correspondan con los tres ejemplos de uso de if que hemos visto.

La expresión dentro de paréntesis es una expresión booleana. **Llamamos expresión booleana a una expresión que solo tiene dos valores posibles:** verdadero (true) o falso (false).

Es importante distinguir la comparación que realizamos con el operador == de la asignación que realizamos con el operador =. Confundirlos nos generará errores de compilación o problemas de lógica en el código. Recuerda que siempre que tengas que comparar con un operador, has de usar == en lugar de =.

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo Rueda (que representa a una rueda de un vehículo). Define una clase Rueda considerando los siguientes atributos de clase: tipo (String), grosor (double), diametro (double), marca (String). Define un constructor asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. Crea un método denominado comprobarDimensiones donde a través de condicionales if realices las siguientes comprobaciones:

- a) Si el diámetro es superior a 1.4 debe mostrarse por consola el mensaje “La rueda es para un vehículo grande”. Si es menor o igual a 1.4 pero mayor que 0.8 debe mostrarse por consola el mensaje “La rueda es para un vehículo mediano”. Si no se cumplen ninguna de las condiciones anteriores debe mostrarse por pantalla el mensaje “La rueda es para un vehículo pequeño”.
- b) Si el diámetro es superior a 1.4 con un grosor inferior a 0.4, ó si el diámetro es menor o igual a 1.4 pero mayor que 0.8, con un grosor inferior a 0.25, deberá mostrarse por consola el mensaje “El grosor para esta rueda es inferior al recomendado”

Compila el código para comprobar que no presenta errores, crea un objeto, usa sus métodos y comprueba que aparezcan correctamente los mensajes por consola. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00637B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONDICIONAL DE SELECCIÓN SWITCH EN JAVA. EJEMPLO DE APLICACIÓN.

La instrucción *switch* es una forma de expresión de un anidamiento múltiple de instrucciones *if ... else*. Su uso no puede considerarse, por tanto, estrictamente necesario, puesto que siempre podrá ser sustituida por el uso de *if*. No obstante, a veces nos resultará útil al introducir mayor claridad en el código.

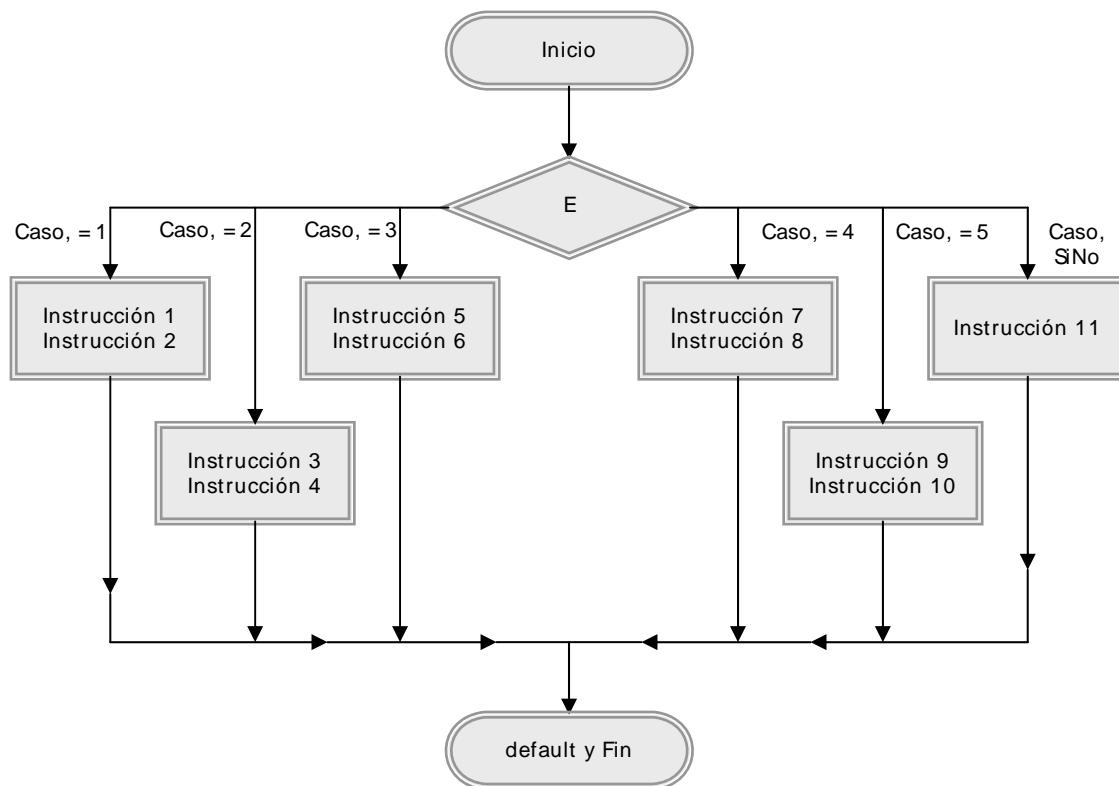


La sintaxis será:

```
switch (expresión) {  
  
    case valor1:  
        instrucciones;  
        break;  
  
    case valor2:  
        instrucciones;  
        break;  
    .  
    .  
    .  
    default:  
        sentencias;  
        break;  
}
```

```
switch (expresión) {  
  
    case valor1:  
    case valor2:  
    case valor3:  
        instrucciones;  
        break;  
  
    case valor4:  
        instrucciones;  
        break;  
    .  
    .  
    .  
    default:  
        sentencias;  
        break;  
}
```

Esquemáticamente a modo de diagrama de flujo:



La cláusula *default* es opcional y representa las instrucciones que se ejecutarán en caso de que no se verifique ninguno de los casos evaluados. El último *break* dentro de un *switch* (en *default* si existe esta cláusula, o en el último caso evaluado si no existe *default*) también es opcional, pero lo incluiremos siempre para ser metódicos.

Switch solo se puede utilizar para evaluar ordinales (por ordinal entenderemos en general valores numéricos enteros o datos que se puedan asimilar a valores numéricos enteros). Por tanto no podemos evaluar cadenas (String) usando switch porque el compilador nos devolverá un error de tipo “*found java.lang.String but expected int*”. Sí se permite evaluar caracteres y lo que se denominan tipos enumerados, que veremos más adelante. Switch solo permite evaluar valores concretos de la expresión: no permite evaluar intervalos (pertenencia de la expresión a un intervalo o rango) ni expresiones compuestas. Código de ejemplo:

```
//Ejemplo de método que usa switch
public void dimeSiEdadEsCritica() {
    switch (edad) {
        case 0:
            System.out.println ("Acaba de nacer hace poco. No ha cumplido el año");
            break;
        case 18: System.out.println ("Está justo en la mayoría de edad"); break;
        case 65: System.out.println ("Está en la edad de jubilación"); break;
        default: System.out.println ("La edad no es crítica"); break;
    }
}
```

En algunos casos escribimos varias instrucciones en una línea y en otros una sola instrucción por línea. Ambas posibilidades son válidas. Prueba a escribir, compilar e invocar este método o uno parecido usando `switch`. Para ello crea primero una clase de nombre Persona cuyos atributos sean nombre y edad. Inicializa los atributos a un valor por defecto en el constructor. Crea métodos para definir valor para los atributos (métodos setters) y prueba el método `dimeSiEdadEsCritica` para comprobar que responde como es de esperar.

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo Motor (que representa el motor de una bomba para mover fluidos). Define una clase Motor considerando los siguientes atributos de clase: `tipoBomba` (int), `tipoFluido` (String), `combustible` (String). Define un constructor asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. Crea un método tipo procedimiento denominado `dimeTipoMotor()` donde a través de un condicional `switch` hagas lo siguiente:

- a) Si el tipo de motor es 0, mostrar un mensaje por consola indicando “No hay establecido un valor definido para el tipo de bomba”.
- b) Si el tipo de motor es 1, mostrar un mensaje por consola indicando “La bomba es una bomba de agua”.
- c) Si el tipo de motor es 2, mostrar un mensaje por consola indicando “La bomba es una bomba de gasolina”.
- d) Si el tipo de motor es 3, mostrar un mensaje por consola indicando “La bomba es una bomba de hormigón”.
- e) Si el tipo de motor es 4, mostrar un mensaje por consola indicando “La bomba es una bomba de pasta alimenticia”.
- f) Si no se cumple ninguno de los valores anteriores mostrar el mensaje “No existe un valor válido para tipo de bomba”.

Compila el código para comprobar que no presenta errores, crea un objeto, usa sus métodos y comprueba que aparezcan correctamente los mensajes por consola. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00638B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

VARIABLES LOCALES A UN MÉTODO O CONSTRUCTOR. SOBRECARGA DE NOMBRES.

Una variable que se declara y se usa dentro de un método (o de un constructor) se dice que es una variable local. Su ámbito es sólo el método o constructor y su tiempo de vida es solo el del método, es decir, son **variables temporales que se crean cuando comienza a ejecutarse el método y se destruyen cuando termina de ejecutarse**.



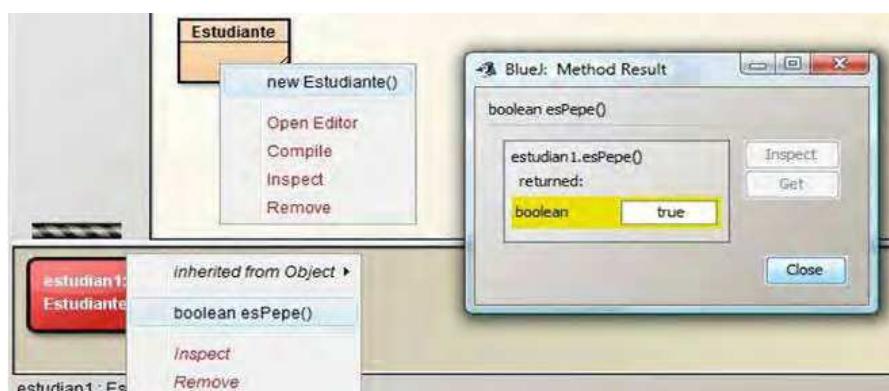
Escribe y compila el siguiente código:

```
public class Estudiante { //El nombre de la clase

    private String nombre; //Campo de los objetos Estudiante

    //Constructor: cuando se cree un objeto Estudiante se ejecutará el código que incluyamos en el constructor
    public Estudiante () {
        nombre = "Pepe";
    } //Cierre del constructor
    //Método que devuelve true si el nombre del objeto tipo Estudiante es Pepe
    public boolean esPepe() {
        boolean seLlamaPepe = false;
        if (nombre == "Pepe") { seLlamaPepe = true; }
        return seLlamaPepe;
    } //Cierre del método
} //Cierre de la clase
```

La variable `seLlamaPepe` es una variable local booleana. Es habitual inicializar las variables locales cuando se las declara, pero no es estrictamente necesario. Si es obligatorio inicializar las variables en algún momento ya que no se debe considerar que tengan un valor por defecto. Crea un objeto de tipo Estudiante pulsando sobre el ícono de la clase con botón derecho y eligiendo la opción `new Estudiante()`. Sobre el objeto que aparecerá en el banco de objetos, invoca el método `esPepe()`.



Ahora crea otro método y trata de establecer en él la variable `seLlamaPepe` con valor `true`. El compilador lanzará un mensaje de error del tipo “*cannot find symbol – variable seLlamaPepe*”. ¿Por qué podemos usar la variable *nombre* en cualquier método mientras que la variable `seLlamaPepe` sólo dentro del método `esPepe()`? Como hemos dicho, el ámbito de una variable declarada en un método es solo ese método. El resto de métodos no conocen la variable. En cambio, un campo de la clase tiene como ámbito toda la clase y lo podemos usar en cualquier lugar del código de la clase. Hay algunas conclusiones interesantes:

- 1) Podemos usar el mismo nombre de variable local en muchos métodos, puesto que no van a interferir entre ellas.
- 2) Una declaración de campo siempre la hacemos precedida de `public` o `private`. En las variables locales, estas palabras clave no se usan debido a su carácter temporal.
- 3) En los métodos tipo función con frecuencia en la sentencia `return` se devuelve como resultado el valor de una variable local que ha sido objeto de cálculo en el método. Tener en cuenta que no se devuelve la variable en sí (que en realidad desaparece cuando termina el método), sino su valor o contenido.

¿Puede un constructor tener variables locales? Sí. Un constructor son una serie de instrucciones. A veces muy sencillas, pero otras veces pueden requerir cálculos o procesos complejos. Por tanto, podemos usar variables locales dentro de ellos declarándolas y usándolas como si se tratara de un método. No tenemos restricciones en cuanto al código que se puede incluir en un constructor.

¿Puede una variable local ser tipo objeto? Sí. Hemos dicho que las clases definen tipos. Por ejemplo podríamos tener una variable local `miTaxi1` declarada como `Taxi miTaxi1;`.

¿Qué ocurre si una variable local tiene el mismo nombre que un campo? Esta situación se daría si tenemos un campo declarado por ejemplo como `String ciudad;` y luego declaramos dentro de un método una variable de ese mismo tipo u otro distinto con el mismo nombre, por ejemplo `boolean ciudad = false;`. En este caso decimos que existe **sobre carga de nombres**. Podemos tener problemas si no manejamos esta situación adecuadamente. Cuando empleemos en el código el nombre de la variable el compilador no es capaz de adivinar nuestro pensamiento para saber si nos referimos al campo de la clase o a la variable local del método. Este conflicto Java lo resuelve aplicando la regla de prevalencia del ámbito “más local”. Es decir, si escribimos un nombre de variable Java usa la variable “más local” disponible. Java tiene prevista la solución para poder usar simultáneamente campos y variables locales con el mismo nombre, mediante el uso de la palabra clave `this`. Esto lo explicaremos más adelante. Reflexionemos ahora sobre los tipos o formas de variables que hemos visto hasta el momento. Se resumen en el siguiente esquema:

		ÁMBITO	DURACIÓN
Tipos de variables	Campos (atributos de objeto)	Toda la clase	Indefinida (vida del objeto)
	Parámetros de un método o constructor	Local solo el método o constructor	Temporal
	Variables locales de un método o constructor	Local solo el método o constructor	Temporal

EJERCICIO

Considera estás desarrollando un programa Java donde necesitas trabajar con objetos de tipo Motor (que representa el motor de una bomba para mover fluidos). Define una clase Motor considerando los siguientes atributos de clase: tipoBomba (int), tipoFluido (String), combustible (String). Define un constructor asignando unos valores de defecto a los atributos y los métodos para poder establecer y obtener los valores de los atributos. Crea un método tipo función que devuelva un booleano (true o false) denominado dimeSiMotorEsParaAgua() donde se cree una variable local booleana motorEsParaAgua de forma que si el tipo de motor tiene valor 1 tomará valor true y si no lo es tomará valor false. El método debe devolver la la variable local booleana motorEsParaAgua.

Compila el código para comprobar que no presenta errores, crea un objeto, usa sus métodos y comprueba que se obtienen resultados correctos. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00639B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CÓMO CREAR CONSTRUCTORES EN JAVA. EJERCICIOS EJEMPLOS RESUELTOS.

Los constructores de una clase son **fragmentos de código que sirven para inicializar un objeto** a un estado determinado. Una clase puede carecer de constructor, pero esto no es lo más habitual. Normalmente todas nuestras clases llevarán constructor. En un constructor es frecuente usar un esquema de este tipo:



```
public MismoNombreQueLaClase (tipo parámetro1, tipo parámetro2 ..., tipo parámetro n ) {
    campo1 = valor o parámetro;
    campo2 = valor o parámetro;
    .
    .
    .
    campo n = valor o parámetro;
}
```

Los constructores tienen el mismo nombre que la clase en la que son definidos y nunca tienen tipo de retorno, ni especificado ni void. Tenemos aquí un aspecto que nos permite diferenciar constructores de métodos: un constructor nunca tiene tipo de retorno mientras que un método siempre lo tiene. Es recomendable que en un constructor se inicialicen **todos** los atributos de la clase aunque su valor vaya a ser nulo o vacío. Si un atributo se quiere inicializar a cero (valores numéricos) siempre lo declararemos específicamente: nombreAtributo = 0;. Si un atributo se quiere inicializar a contenido nulo (atributos que son objetos) siempre lo declararemos específicamente: nombreAtributo = null;. Si un atributo tipo texto se quiere inicializar vacío siempre lo declararemos específicamente: nombreAtributo = “”;. El motivo para actuar de esta manera es que declarando los atributos como nulos o vacíos, dejamos claro que esa es nuestra decisión como programadores. Si dejamos de incluir uno o varios campos en el constructor puede quedar la duda de si hemos olvidado inicializar ese campo o inducir a pensar que trabajamos con malas prácticas de programación.

La inicialización de campos y variables es un proceso muy importante. Su mala definición es fuente de problemas en el desarrollo de programas. Como regla de buena programación, cuando crees campos o variables, procede de forma inmediata a definir su inicialización.

Un constructor puede:

- Carecer de parámetros: que no sea necesario pasarle un parámetro o varios al objeto para inicializarse. Un constructor sin parámetros se denomina “constructor general”.

- b) Carecer de contenido. Por ejemplo, `public Taxi () {}` podría ser un constructor, vacío. En general un constructor no estará vacío, pero en algunos casos particulares puede estarlo. Si el constructor carece de contenido los campos se inicializan con valor nulo o, si son tipos definidos en otra clase, como se haya definido en el constructor de la otra clase. Excepto en casos controlados, evitaremos que existan constructores vacíos.

Si un constructor tiene parámetros, el funcionamiento es análogo al que ya hemos visto para métodos. Cuando vayamos a crear el objeto con BlueJ, se nos pedirá además del nombre que va a tener el objeto, el valor o contenido de los parámetros requeridos. Un parámetro con frecuencia sirve para inicializar el objeto como hemos visto, y en ese caso el objeto tendrá el valor pasado como parámetro como atributo “para siempre”, a no ser que lo cambiemos por otra vía como puede ser un método modificador del atributo. No obstante, en algunos casos los parámetros que recibe un constructor no se incorporarán directamente como atributos del objeto sino que servirán para realizar operaciones de diversa índole. Escribe y compila el siguiente código:

```
public class Taxi { //El nombre de la clase

    private String ciudad; //Ciudad de cada objeto taxi
    private String matricula; //Matrícula de cada objeto taxi
    private String distrito; //Distrito asignado a cada objeto taxi
    private int tipoMotor; //Tipo de motor asignado a cada objeto taxi. 0 = desconocido, 1 = gasolina, 2 = diesel

    //Constructor: cuando se cree un objeto taxi se ejecutará el código que incluyamos en el constructor
    public Taxi (String valorMatricula, String valorDistrito, int valorTipoMotor) {
        ciudad = "México D.F.";
        matricula = valorMatricula;
        distrito = valorDistrito;
        tipoMotor = valorTipoMotor;
    } //Cierre del constructor

    //Método para obtener la matrícula del objeto taxi
    public String getMatricula () { return matricula; } //Cierre del método

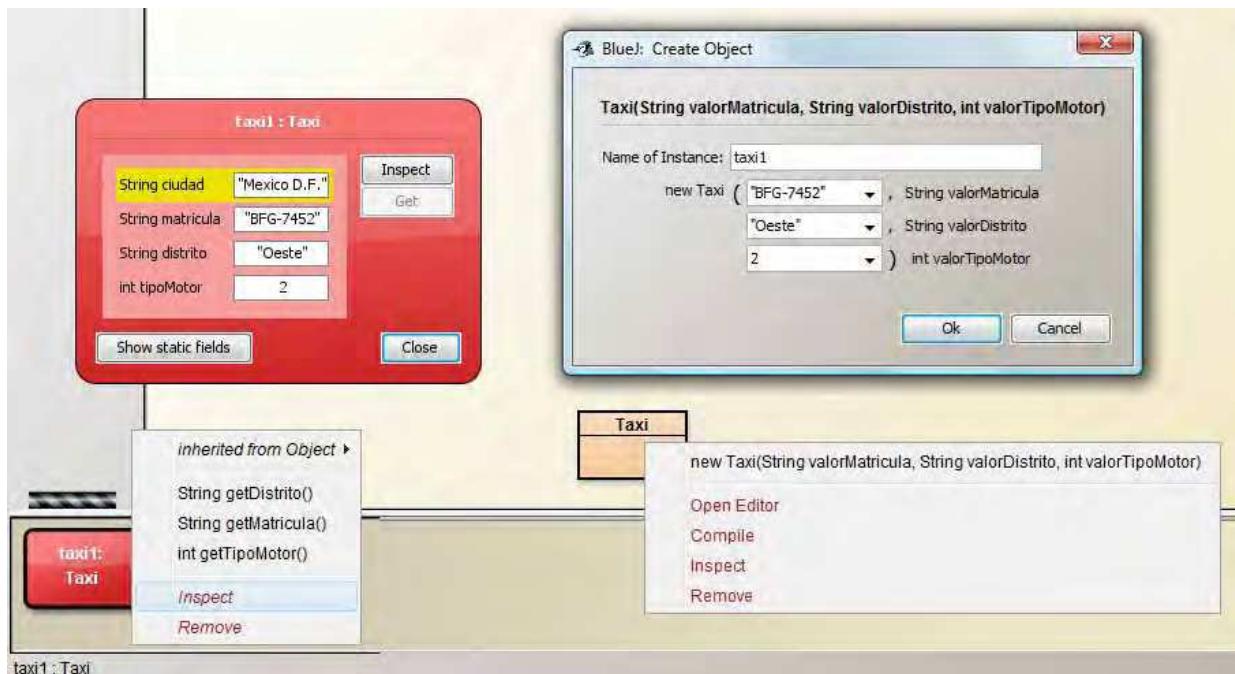
    //Método para obtener el distrito del objeto taxi
    public String getDistrito () { return distrito; } //Cierre del método

    //Método para obtener el tipo de motor del objeto taxi
    public int getTipoMotor () { return tipoMotor; } //Cierre del método

} //Cierre de la clase
```

Este código es similar al que vimos en epígrafes anteriores. La diferencia radica en que ahora en vez de tener un constructor que establece una forma fija de inicializar el objeto, **la inicialización depende de los parámetros que le lleguen al constructor**. Además, hemos eliminado los métodos para establecer el valor de los atributos. Ahora éstos solo se pueden consultar mediante los métodos `get`. Pulsa con botón derecho sobre el ícono de la clase y verás como la opción `new Taxi` incluye ahora los parámetros dentro de los paréntesis. Escoge esta opción y establece unos valores como matrícula “BFG-7452”, distrito

“Oeste” y tipo de motor 2. Luego, con el botón derecho sobre el ícono del objeto, elige la opción *Inspect* para ver su estado.



Que un constructor lleve o no parámetros y cuáles tendremos que elegirlo para cada clase que programemos. En nuestro ejemplo hemos decidido que aunque la clase tiene cuatro campos, el constructor lleve solo tres parámetros e inicializar el campo restante con un valor fijo. Un constructor con parámetros es adecuado si tiene poco sentido inicializar los objetos vacíos o siempre con el mismo contenido para uno o varios campos. No obstante, **siempre hay posibilidad de darle contenido a los atributos a posteriori si incluimos métodos “setters”**. El hacerlo de una forma u otra dependerá del caso concreto al que nos enfrentemos.

El esquema que hemos visto supone que en general vamos a realizar una declaración de campo en cabecera de la clase, por ejemplo `String ciudad;`, y posteriormente inicializar esa variable en el constructor, por ejemplo `ciudad = "Méjico D.F.;"`. ¿Sería posible hacer una declaración en cabecera de clase del tipo `String ciudad = "Méjico D.F.;"`? La respuesta es que sí. El campo quedaría inicializado en cabecera, pero esto en general debe ser considerado una mala práctica de programación y contraproducente dentro de la lógica de la programación orientada a objetos. Por tanto de momento trataremos de evitar incluir código de ese tipo en nuestras clases y procederemos siempre a inicializar en los constructores.

EJERCICIO

Define una clase Bombero considerando los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), especialista (boolean). Define un constructor que reciba los parámetros necesarios para la inicialización y los métodos para poder establecer y obtener los valores de los atributos. Compila el código para comprobar que no presenta errores, crea un objeto y comprueba que se inicializa correctamente consultando el valor de sus atributos después de haber creado el objeto. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00640B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CLASES CON DOS O MÁS CONSTRUCTORES. SOBRECARGA DE CONSTRUCTORES O MÉTODOS.

En este apartado vamos a ver cómo una clase en Java puede tener más de un constructor y a entender qué implicaciones y significado tiene esto. Escribiremos el código de una clase y lo compilaremos para ir analizando en base a este código cómo se generan clases con varios constructores y el significado del concepto de sobrecarga de constructores o métodos.



Escribe y compila el siguiente código:

```
//Ejemplo de clase con dos constructores y un método
public class Persona {
    private String nombre;
    private int edad;

    public Persona (String nombrePersona) { //CONSTRUCTOR 1
        nombre = nombrePersona;
        edad = 0;    }

    public Persona () { //CONSTRUCTOR2
        nombre = "";
        edad = 0;    }

    public String getNombre () { return nombre; } //Cierre del método
} //Cierre de la clase
```

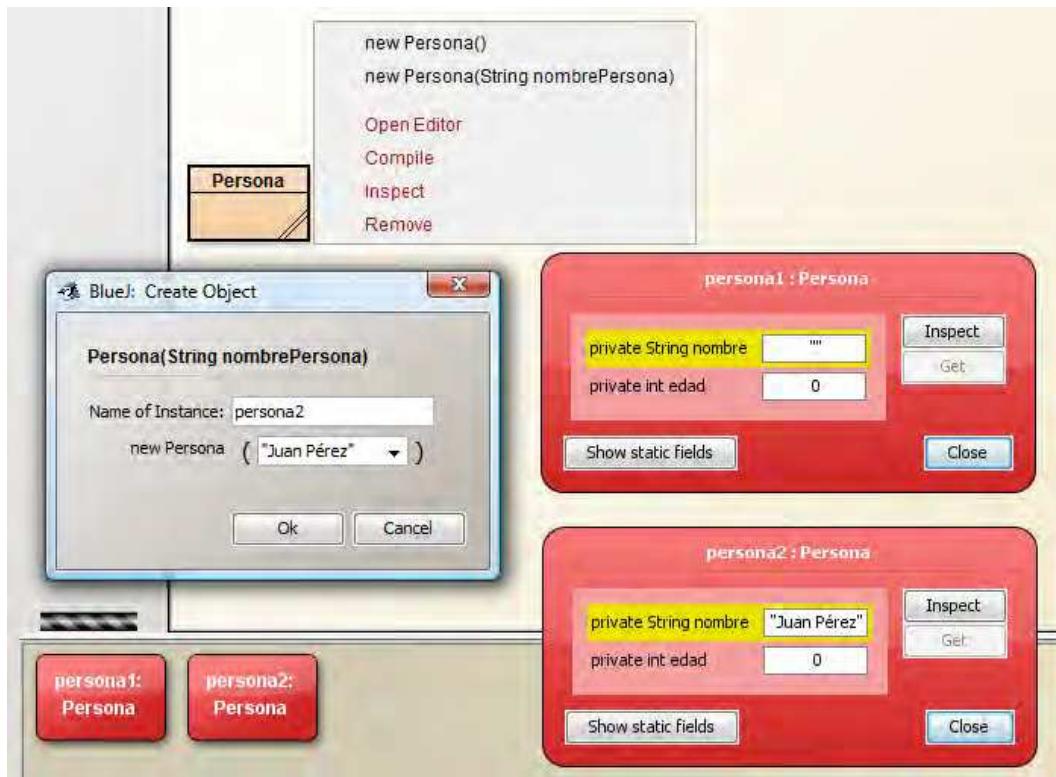
Hemos definido una clase, denominada *Persona*, que nos permite crear objetos de tipo *Persona*. Todo objeto de tipo *Persona* estará definido por dos campos: *nombre* (tipo *String*) y *edad* (tipo entero), y admitirá un método: *getNombre()*. Al realizar la invocación *nombreDelObjeto.getNombre()* obtendremos el atributo *nombre* del objeto.

La clase tiene dos constructores. ¿Qué significado tiene esto? Pues que **podremos crear personas de dos maneras distintas**:

- Personas que se creen con el constructor 1: habrá de indicarse, además del nombre del objeto, el parámetro que transmite el nombre de la persona.
- Personas que se creen con el constructor 2: no requieren parámetros para su creación y se inicializan a unos valores por defecto (nombre cadena vacía y edad cero).

Cuando más de un constructor o método tienen el mismo nombre pero distintos parámetros decimos que el constructor o método está **sobrecargado**. La sobrecarga de constructores o métodos permite llevar a cabo una tarea de distintas maneras (por ejemplo crear un objeto *Persona* con un nombre ya establecido o crearlo sin nombre establecido).

Pulsa sobre el icono de la clase y elige la opción `new Persona()` para crear un objeto. Seguidamente, pulsa de nuevo sobre el icono de la clase y con botón derecho elige la opción correspondiente al otro constructor disponible `new Persona(String nombrePersona)`. Introduce un nombre como parámetro, por ejemplo “Juan Pérez”. A continuación, utiliza el inspector de objetos para comprobar cuál es el estado de cada objeto.



La existencia de dos constructores se ha visto reflejada en que disponemos de más de una opción de `new Persona` para crear objetos. Según la opción que elijamos, el objeto Persona se creará de una forma u otra. Esto nos lleva a la conclusión de que cada constructor define una forma de crear objetos.

EJERCICIO

Define una clase Profesor considerando los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), especialista (boolean). Define un constructor que reciba los parámetros necesarios para la inicialización y otro constructor que no reciba parámetros. Crea los métodos para poder establecer y obtener los valores de los atributos. Compila el código para comprobar que no presenta errores, crea un objeto usando un constructor y luego otro objeto usando el otro constructor. Comprueba que se inicializan correctamente consultando el valor de sus atributos después de haber creado los objetos. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00641B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CLASES QUE UTILIZAN OBJETOS. RELACIÓN DE USO ENTRE CLASES. DIAGRAMAS DE CLASES.

Hemos visto hasta ahora clases que definen tipos donde los campos son variables de tipo primitivo o String. Analicemos ahora la posibilidad de crear clases donde los atributos sean tipos que hayamos definido en otras clases.



Consideraremos que partimos de las clases Taxi y Persona ya escritas y compilando correctamente conforme a los ejemplos vistos en epígrafes anteriores.



Escribe y compila el siguiente código:

```

//Ejemplo de clase que utiliza tipos definidos en otras clases (usa otras clases)
public class TaxiCond {

    private Taxi vehiculoTaxi;
    private Persona conductorTaxi;

    //Constructor
    public TaxiCond () {
        vehiculoTaxi = new Taxi (); //Creamos un objeto Taxi con el constructor general de Taxi
        conductorTaxi = new Persona (); //Creamos un objeto Persona con el constructor general de Persona
    }

    public void setMatricula (String valorMatricula) { vehiculoTaxi.setMatricula(valorMatricula); }

    //Método que devuelve la información sobre el objeto TaxiCond
    public String getDatosTaxiCond () {
        String matricula = vehiculoTaxi.getMatricula();
        String distrito = vehiculoTaxi.getDistrito();
        int tipoMotor = vehiculoTaxi.getTipoMotor();
        String cadenaTipoMotor = "";
    }
}
  
```

```

if (tipoMotor ==0) { cadenaTipoMotor = "Desconocido"; }
else if (tipoMotor == 1) { cadenaTipoMotor = "Gasolina"; }
else if (tipoMotor == 2) { cadenaTipoMotor = "Diesel"; }

String datosTaxiCond = "El objeto TaxiCond presenta estos datos. Matrícula: " + matricula +
    " Distrito: " + distrito + " Tipo de motor: " + cadenaTipoMotor;

System.out.println (datosTaxiCond);
return datosTaxiCond;
} //Cierre del método

} //Cierre de la clase

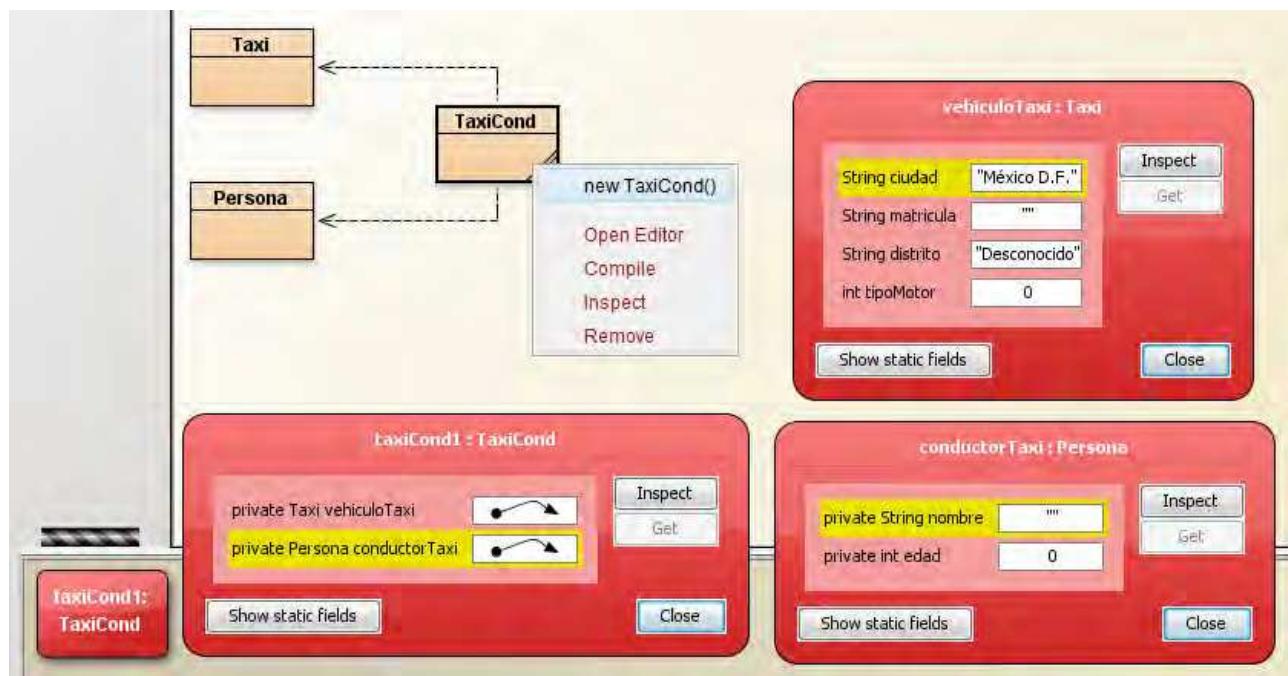
```

Analicemos ahora lo que hace este código. Creamos una clase denominada `TaxiCond` (que podemos interpretar como “taxi con conductor”). Los objetos del tipo `TaxiCond` decimos que van a constar de dos campos: un objeto `Taxi` y un objeto `Persona`. Fíjate que estamos utilizando el nombre de otra clase como si fuera el tipo de una variable “normal y corriente”. Esto es posible porque las clases definen tipos. Desde el momento en que nuestra clase utiliza tipos definidos por otras clases decimos que se establece una relación de uso: `TaxiCond` usa a `Taxi` y a `Persona`. El constructor de `TaxiCond` inicializa los objetos de este tipo para que consten de un objeto `Taxi` creado con el constructor por defecto y de una `Persona` creada con el constructor por defecto.

Para los objetos de tipo `TaxiCond` hemos definido dos métodos (podríamos haber definido muchos más) que son: el método modificador y con parámetros `setMatricula(String valorMatricula)` y el método observador y sin parámetros `getDatosTaxiCond()`. Un aspecto muy importante del código es que desde el momento en que usamos objetos en una clase, podemos acceder a los métodos públicos propios de esos objetos cuyo código se encontrará en otra clase. Por ejemplo la invocación `vehiculoTaxi.setMatricula(valorMatricula);` llama un método propio de los objetos `Taxi` que se encuentra definido en otra clase. Es decir, todo objeto puede llamar a sus métodos públicos independientemente de dónde se encuentre.

Una vez compilado el código, en el diagrama de clases se nos muestran unas flechas discontinuas que relacionan la clase `TaxiCond` con las clases `Taxi` y `Persona`. Estas flechas discontinuas lo que indican es que hay una relación de uso entre las clases. En algunas circunstancias BlueJ puede mantener erróneamente indicadores de relación que no son ciertos. En estos casos, las flechas pueden eliminarse seleccionándolas y con botón derecho eligiendo la opción Remove. También pueden crearse eligiendo el botón ----> en la parte superior izquierda de la pantalla y a continuación pulsando primero el icono de la “clase que usa” y luego el icono de la “clase que es usada”.

Crea un objeto de tipo `TaxiCond` pulsando sobre el icono de la clase y con botón derecho eligiendo new `TaxiCond()`. A continuación con botón derecho sobre el objeto elige la opción Inspect. La ventana que se nos muestra nos indica que el objeto consta de dos campos, pero en el recuadro correspondiente al valor de dichos campos en vez de un valor nos aparece una flecha curvada. Esta flecha lo que nos indica es que el campo no contiene un valor simple (como un entero) sino un objeto. La flecha **simboliza una referencia al objeto, ya que el objeto no se puede representar directamente** al ser una entidad compleja.



Pulsa sobre la flecha de referencia de cada uno de los campos y luego sobre el botón **Inspect** de la ventana. Se te abrirán otras dos ventanas donde puedes observar los valores de los campos de cada uno de los objetos que forman el objeto **TaxiCond**. Pero ten en cuenta que un objeto siempre podría tener como campo otro objeto, es decir, podríamos seguir observando “flechas” una y otra vez al ir inspeccionando objetos y esto sería una situación normal. Estamos trabajando con programación orientada a objetos, por tanto que aparezcan objetos “por todos lados” será normal.

La relación de uso entre clases es una de los tipos de relación más habituales en programación orientada a objetos. Las variables de instancia de un objeto pueden ser tanto de tipo primitivo como tipo objeto. Recordar que la variable que define un objeto no contiene al objeto en sí mismo, sino una referencia al espacio de memoria donde se encuentra. Dado que un objeto es una entidad compleja simbólicamente se representa con una línea que comienza en un punto y termina en una punta de flecha. Un objeto puede crearse e invocar sus métodos públicos desde distintas clases y decimos que esto establece una relación de uso entre clases. Por tanto, el código fuente de una clase puede ser usado desde otras clases.

Un esquema donde se representan las clases y las relaciones que existen entre ellas se denomina **diagrama de clases** y nos sirve para comprender la estructura de los programas. Se dice que el diagrama de clases constituye una vista estática del programa, porque es un esquema fijo de relaciones dentro del programa. Sin embargo, el que exista una relación entre clases no significa que en un momento dado vaya a existir un objeto que materialice esa relación entre clases. Es posible que el programa comience y que pase un tiempo antes de que se cree un objeto que refleje la relación entre

clases. Si representáramos los objetos existentes en un momento dado y las relaciones entre ellos tendríamos una vista dinámica del programa. El inconveniente de las vistas dinámicas es que los objetos se crean, destruyen o cambian sus relaciones continuamente, por lo que representarlas resulta costoso. Por este motivo, no utilizaremos las vistas dinámicas. Sin embargo, sí usaremos con frecuencia los diagramas de clases para comprender la estructura de nuestro código.

Nos queda una aclaración por realizar: ¿Por qué si los tipo String son objetos BlueJ nos informa directamente de su contenido en vez de mostrar una flecha? La razón para ello estriba en que el tipo String es un objeto un tanto especial, ya que su contenido es relativamente simple comparado con el de otros objetos. Para facilitar el trabajo BlueJ nos informa directamente de su contenido, pero no podemos olvidar que un String es un objeto y esto tiene una relevancia notable como veremos más adelante.

Para completar la comprensión de la relación de uso entre clases, utiliza los métodos disponibles para el objeto TaxiCond que has creado: establece distintos valores de matrícula con el método setMatricula y visualiza los datos del objeto con el método getDatosTaxiCond. Crea además nuevos métodos que te permitan establecer el distrito y el tipo de motor de los objetos TaxiCond. Te planteamos otra reflexión: al igual que hemos definido un tipo TaxiCond que tiene dos objetos como campos, podemos definir tipos que tengan cinco, diez, quince o veinte objetos como campos. Por ejemplo, un objeto Casa podría definirse con estos campos:

```
private Salon salonCasa;  
private Cocina cocinaCasa;  
private Baño baño1Casa;  
private Baño baño2Casa;  
private Jardin jardinCasa;  
private Dormitorio dormitorio1Casa;  
private Dormitorio dormitorio2Casa;  
private Dormitorio dormitorio3Casa;
```

Ten en cuenta que **dentro de un objeto puedes tener n objetos de otro tipo**. En este ejemplo, dentro de un objeto Casa tenemos dos objetos de tipo Baño y tres objetos de tipo Dormitorio. Todos los objetos Dormitorio van a tener los mismos atributos y métodos, pero cada objeto tendrá su propio estado en cada momento.

EJERCICIO

Define tres clases: Casa, SalonCasa y CocinaCasa. La clase SalonCasa debe tener como atributos numeroDeTelevisores (int) y tipoSalon (String) y disponer de un constructor que los inicialice a 0 y "desconocido". La clase CocinaCasa debe tener como atributos esIndependiente (boolean) y numeroDeFuegos (int) y un constructor que los inicialice a false y 0. La clase Casa tendrá los siguientes

atributos de clase: superficie (double), direccion (String), salonCasa (tipo SalonCasa) y cocina (tipo CocinaCasa). Define un constructor para la clase Casa que establezca a unos valores de defecto los atributos simples y que cree nuevos objetos si se trata de atributos objeto. Compila el código para comprobar que no presenta errores, crea un objeto de tipo Casa. Comprueba que se inicializan correctamente consultando el valor de sus atributos después de haber creado los objetos. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00642B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PASO DE OBJETOS COMO PARÁMETROS A UN MÉTODO O CONSTRUCTOR EN JAVA

Hasta ahora habíamos visto como un método o constructor puede recibir parámetros de los tipos de datos primitivos en Java, como int, boolean, etc. e incluso de tipos envoltorio como Integer. Vamos a ver que también se pueden recibir otro tipo de parámetros. Partimos de esta definición de clase Taxi, escríbela en tu editor:



```
/* Esta clase representa un taxi ejemplo - aprenderaprogramar.com */

public class Taxi { //El nombre de la clase
    private String ciudad; //Ciudad de cada objeto taxi
    private String matricula; //Matrícula de cada objeto taxi
    private String distrito; //Distrito asignado a cada objeto taxi
    private int tipoMotor; //tipo de motor asignado a cada objeto taxi. 0 = desconocido, 1 = gasolina, 2 = diesel

    //Constructor 1: constructor sin parámetros
    public Taxi () {
        ciudad = "México D.F.";    matricula = "";    distrito = "Desconocido";    tipoMotor = 0;
    } //Cierre del constructor

    //Constructor 2: constructor con parámetros
    public Taxi (String valorMatricula, String valorDistrito, int valorTipoMotor) {
        ciudad = "México D.F.";    matricula = valorMatricula;    distrito = valorDistrito;    tipoMotor = valorTipoMotor;
    } //Cierre del constructor

    //Método para establecer la matrícula de un taxi
    public void setMatricula (String valorMatricula) { matricula = valorMatricula; } //Cierre del método
    //Método para establecer el distrito de un taxi
    public void setDistrito (String valorDistrito) { distrito = "Distrito " + valorDistrito; } //Cierre del método
    //Método para establecer el tipo de motor de un taxi
    public void setTipoMotor (int valorTipoMotor) { tipoMotor = valorTipoMotor; } //Cierre del método

    //Método para obtener la matrícula del objeto taxi
    public String getMatricula () { return matricula; } //Cierre del método

    //Método para obtener el distrito del objeto taxi
    public String getDistrito () { return distrito; } //Cierre del método

    //Método para obtener el tipo de motor del objeto taxi
    public int getTipoMotor () { return tipoMotor; } //Cierre del método
} //Cierre de la clase
```

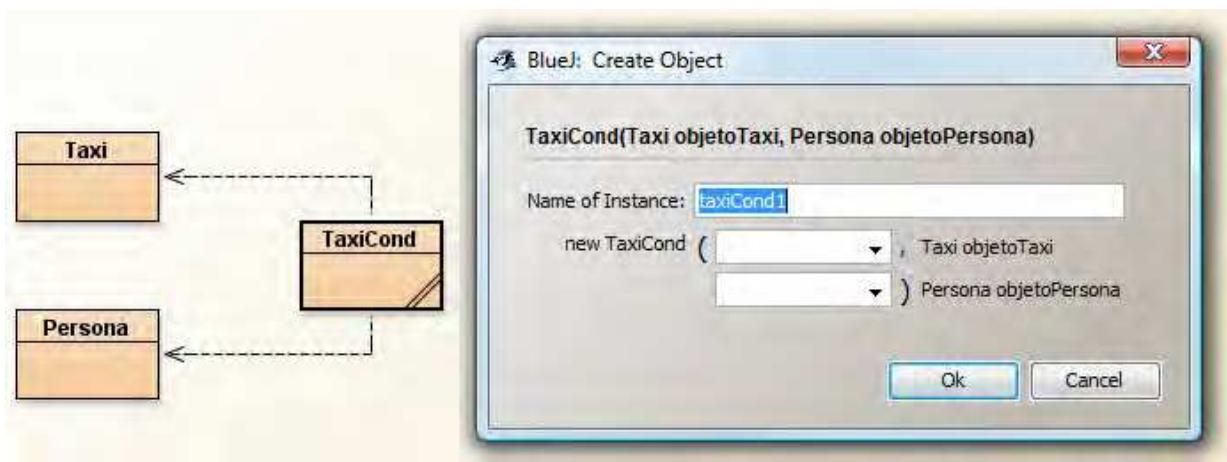
Recupera el código de la clase Persona que usamos anteriormente y crea la clase Persona. Modifica el código de la clase TaxiCond que usamos anteriormente de forma que el constructor pase a ser este:

```
//Constructor
public TaxiCond (Taxi objetoTaxi, Persona objetoPersona) {
    //Creamos un objeto Taxi con los mismos datos del Taxi recibido como parámetro
    vehiculoTaxi = new Taxi (objetoTaxi.getMatricula(), objetoTaxi.getDistrito(), objetoTaxi.getTipoMotor() );
    //Creamos un objeto Persona con los mismos datos de la Persona recibidos como parámetro
    conductorTaxi = new Persona (objetoPersona.getNombre() ); }
```

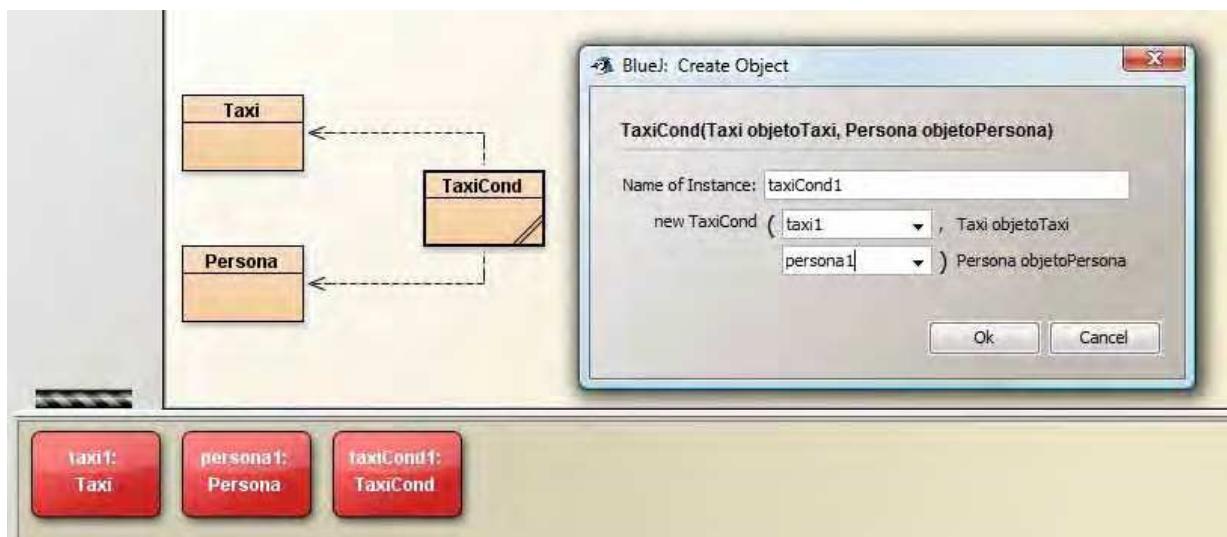
Ahora debes tener tres clases: Taxi, Persona y TaxiCond. La clase Taxi representa un taxi, la clase Persona representa una persona y la clase TaxiCond representa un taxi con conductor.

¿Por qué hemos usado una expresión como `vehiculoTaxi = new Taxi (objetoTaxi.getMatricula(), objetoTaxi.getDistrito(), objetoTaxi.getTipoMotor());` en vez de simplemente `vehiculoTaxi = objetoTaxi;`? La respuesta hay que buscarla en algo que tendremos que analizar más ampliamente un poco más adelante: **un objeto es algo distinto a un tipo primitivo y no podemos aplicarle la lógica de los tipos primitivos.** El hecho de crear nuevos objetos responde a que no queremos modificar los objetos que se pasan como parámetro. Esto entra dentro de la lógica y propiedades de los objetos que iremos estudiando poco a poco. De momento, simplemente utilizaremos ahora este código sin pararnos a pensar demasiado en él.

Nos encontramos frente a un constructor que nos requiere como parámetros objetos complejos y no tipos primitivos ni objetos simples de tipo String. Intenta ahora crear un objeto de tipo TaxiCond pulsando sobre el ícono de la clase y con botón derecho eligiendo la opción new TaxiCond.



Aparecerá una ventana que nos pide además del nombre de la instancia (objeto que vamos a crear), que indiquemos los dos objetos necesarios para crear el nuevo objeto. Nosotros no disponemos de esos dos objetos, así que vamos a cerrar esta ventana y a crear primero un objeto Taxi y un objeto Persona, que nos van a ser necesarios para crear el objeto TaxiCond. Luego, crea el objeto TaxiCond introduciendo los nombres de los objetos Taxi y Persona creados previamente en las casillas correspondientes (también resulta válido hacer click sobre los iconos de los objetos).



Con este ejemplo hemos comprobado que un constructor (o un método) nos puede requerir como parámetros uno o varios objetos y que para pasar los mismos escribimos sus nombres.

EJERCICIO

Define tres clases: Casa, SalonCasa y CocinaCasa. La clase **SalonCasa** debe tener como atributos **numeroDeTelevisores** (int) y **tipoSalon** (String) y disponer de un constructor que los inicialice a 0 y "desconocido". La clase **CocinaCasa** debe tener como atributos **esIndependiente** (boolean) y **numeroDeFuegos** (int) y un constructor que los inicialice a false y 0. La clase **Casa** tendrá los siguientes atributos de clase: **superficie** (double), **direccion** (String), **salonCasa** (tipo **SalonCasa**) y **cocina** (tipo **CocinaCasa**). Define un constructor para la clase **Casa** que establezca a unos valores de defecto los atributos simples y que cree nuevos objetos si se trata de atributos objeto. Define otro constructor que reciba como parámetros la superficie, dirección y un objeto de tipo **SalonCasa** y otro de tipo **CocinaCasa**. Compila el código para comprobar que no presenta errores, y crea un objeto de tipo **Casa** usando el constructor que recibe parámetros. Ten en cuenta que antes tendrás que haber creado los objetos de tipo **SalonCasa** y **CocinaCasa** para poder pasárselos al constructor. Comprueba que el objeto **Casa** se inicializa correctamente consultando el valor de sus atributos después de haber creado el objeto. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00643B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

LA SENTENCIA NEW COMO INVOCACIÓN DE UN CONSTRUCTOR EN JAVA

En el ejemplo anterior hemos visto cómo usar el entorno de desarrollo BlueJ para crear objetos en Java. La forma de creación de objetos ha sido a través del IDE y con una visualización gráfica. Si escribiéramos el código correspondiente a lo que hemos hecho en el ejemplo anterior usando los iconos de BlueJ podría ser algo así:



```
Taxi taxi1 = new Taxi(); //Creación de un objeto tipo Taxi
Persona persona1 = new Persona(); //Creación de un objeto tipo Persona
TaxiCond taxiCond1 = new TaxiCond (taxi1, persona1); /*Creación de un objeto tipo
TaxiCond pasando como parámetros otros objetos creados previamente*/
```

Tener en cuenta que cuando incluimos como atributo de una clase un objeto usando una sintaxis del tipo: *private NombreDeLaOtraClase nombreDelObjeto;*, con esta declaración estamos creando la variable apuntadora (referencia) al objeto, pero **el objeto en sí mismo todavía no se ha creado.** La creación del objeto en código se indica usando esta sintaxis:

nombreDelObjeto = new NombreDeLaOtraClase (parámetros requeridos por el constructor de la otra clase si los hubiera);

Recordar que la variable *nombreDelObjeto* contiene una referencia (puntero) al objeto, no el objeto en sí mismo. La instrucción *new* implica que se producen dos acciones:

- Creación de un objeto de un tipo definido por una clase.
- Ejecución del constructor asociado.

Si una clase define varios constructores, el constructor invocado por la sentencia *new* es el que coincide en número y tipo de parámetros con los utilizados en la sentencia *new*. Por ejemplo: *taxi1 = new Taxi();* invoca al constructor general, mientras que *taxi1 = new Taxi ("BFG-7432")* invoca al constructor que requiere un String como parámetro. *new Taxi ("BFG-7432", "Oeste")* invocaría al constructor que requiere dos String como parámetros. *new Taxi ("BFG-7432", "Oeste", 2)* invocaría al constructor que requiere dos String y un entero como parámetros, etc. No puede haber dos constructores que requieran el mismo número y tipo de parámetros (por ejemplo dos constructores que requieran un String) porque eso generaría una ambigüedad que daría lugar a un error de compilación.

Próxima entrega: CU00644B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

LA CLASE VISTA COMO PAQUETE DE CÓDIGO. OBJETOS DEL MUNDO REAL Y ABSTRACTOS.

Cuando hablamos de operadores aritméticos en Java planteamos que aunque en otros lenguajes existe un operador de exponenciación, en Java no es así. Para calcular una potencia dijimos que podíamos hacer varias cosas:

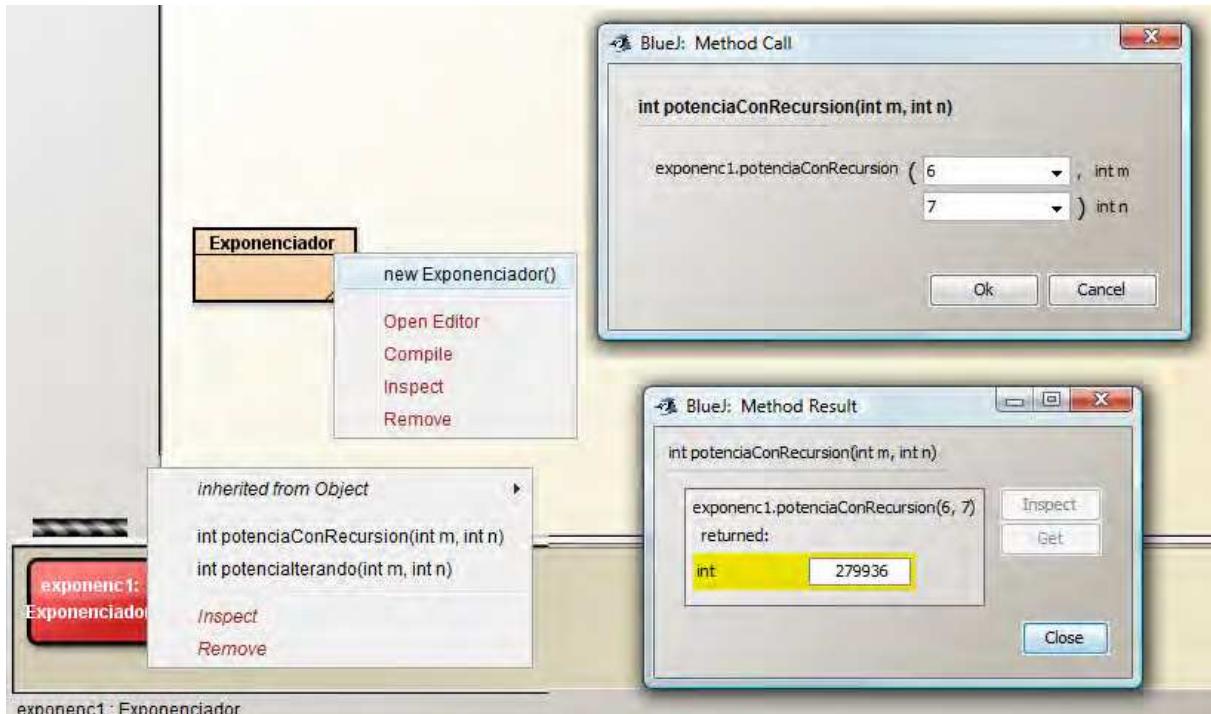


- a) Recurrir a multiplicar n veces el término. Por ejemplo \min^3 lo podemos calcular como $\min * \min * \min$. Obviamente esto no es práctico para potencias de exponentes grandes.
- b) Usar un bucle que dé lugar a la repetición de la operación multiplicación n veces, o usar un método que ejecute la operación.
- c) Usar herramientas propias del lenguaje que permiten realizar esta operación.

Podemos crear nuestros propios métodos para realizar una exponenciación. Este podría ser el código para ello:

```
//Clase que permite elevar un número entero m a otro número entero n y obtener un resultado
public class Exponenciador {
    //Constructor
    public Exponenciador () {
        //Nada que declarar
    }
    //Método 1 para calcular la potencia
    public int potenciaIterando (int m, int n) {
        int resultado = 1;
        for (int i=1; i<=n; i++) {
            resultado = resultado * m;
        }
        return resultado;
    } //Cierre del método
    //Método 2 para calcular la potencia
    public int potenciaConRecursion (int m, int n) {
        if (n==0) { return 1;
        } else { return m * potenciaConRecursion (m, n-1); }
    } //Cierre del método
} //Cierre de la clase
```

Escribe y compila el código anterior sin preocuparte de si lo entiendes completamente o no. Seguidamente, crea un objeto Exponenciador e invoca sus métodos *potencialterando* y *potenciaConRecursion* pasando como parámetros 2 y 3. El resultado en ambos casos debe ser 8, que es el resultado de ejecutar $2^3 = 2 * 2 * 2$. Haz lo mismo pasando como parámetros 6 y 7. El resultado debe ser 279936, que es el valor que se obtiene al ejecutar 6^7 .



Vamos a repasar lo que hemos hecho: hemos definido una clase sin campos, es decir, un objeto de tipo Exponenciador no va a tener atributos. **¿Por qué no tiene atributos?** Simplemente porque no los necesitamos. Esa clase permite crear objetos de tipo Exponenciador sobre los que se pueden invocar dos métodos que al fin y al cabo hacen lo mismo (obtener una potencia) pero de distinta manera. El método iterativo es más fácil de entender: repetimos una multiplicación n veces. El método recursivo es más difícil de entender: se basa en que un método se llame a sí mismo repetidamente cambiando los parámetros de la llamada (n va disminuyendo una unidad en cada llamada) hasta que se llega a un caso “terminal” denominado caso base.

La recursión es una forma de programación que podríamos denominar avanzada, así que no te preocupes ahora de entenderla. Hay programadores con muchos años de experiencia que no la usan directamente porque el seguimiento de procesos recursivos es en general complicado. Ahora simplemente quédate con la idea de que una misma cosa se puede hacer de distintas maneras, y que esas maneras pueden tener distintas propiedades (por ejemplo una ser más rápida que otra, una consumir más memoria que otra, una ser más difícil de seguir que otra, etc.).

Podríamos haber usado otro diseño para lo que hemos hecho: en vez de tener una clase Exponenciador con dos métodos, podríamos haber definido dos clases denominadas ExponenciadorIterativo y ExponenciadorRecursivo, cada una con un método. ¿Es preferible hacerlo de una manera o de otra? La respuesta es que depende. Cuando trabajamos programando tenemos que tomar decisiones de diseño

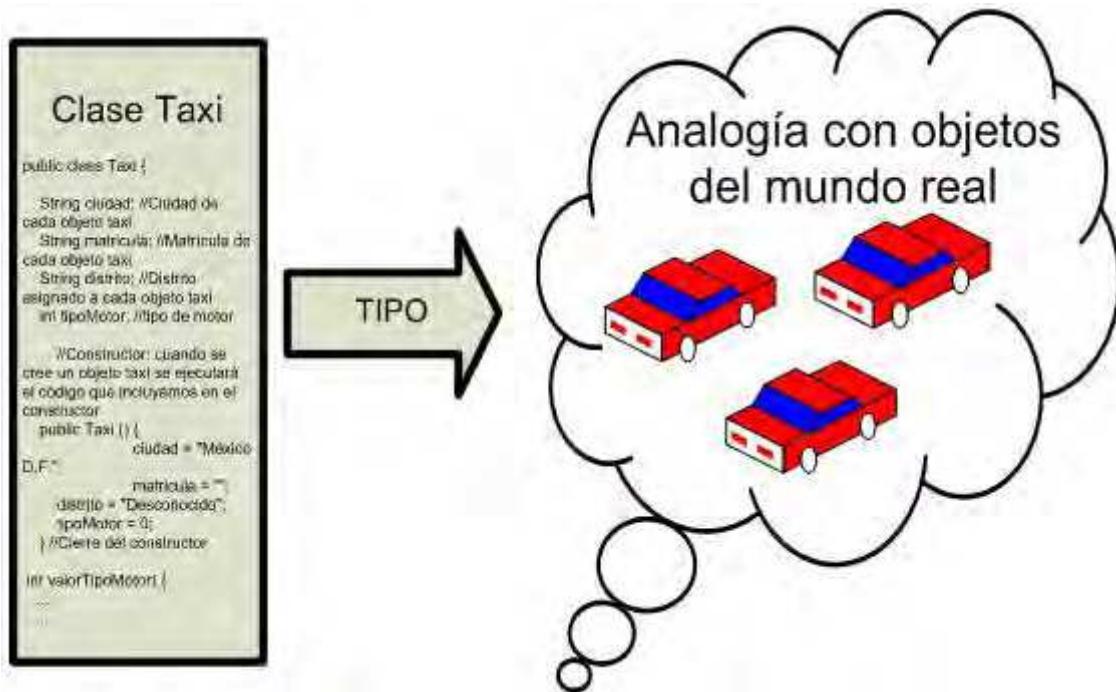
que a veces son más fáciles de tomar y a veces más difíciles. Estas decisiones de diseño pueden tener en algunos casos poca relevancia, pero en otros muchos importantes repercusiones en el futuro del desarrollo de los programas. Conviene analizar las opciones con detenimiento y valorar cuál es la más apropiada. En nuestro caso se trata solo de un ejemplo y elegir una opción u otra nos resulta digamos que indiferente.

Hay algunas cuestiones que merece la pena comentar del código que hemos escrito.

¿Por qué el constructor de esta clase está vacío? Cuando definimos *clase* dijimos que una clase es una abstracción que define un tipo de objeto especificando qué propiedades (atributos) y operaciones disponibles va a tener. Además habíamos dicho que en general un constructor no estará vacío. Llegados a este punto no queda más que buscar otra definición de clase:

Clase: es un paquete o fragmento de código Java que permite crear al menos una instancia (objeto).

En nuestro ejemplo de clase Taxi, el tipo definido por la clase tenía una correspondencia con el mundo real o con objetos que podíamos concebir físicamente.



En nuestro ejemplo de clase Exponenciador, el tipo definido por la clase podemos preguntarnos si tiene alguna analogía en objetos que podamos identificar en el mundo real.



La respuesta es que en este caso no podemos encontrar una analogía. Efectivamente, podemos ir por la calle y encontrarnos un Taxi pero difícilmente iremos por la calle y nos encontraremos un Exponenciador. Aquí radica una dificultad habitual que tienen las personas que están aprendiendo Java. Cuando existe una analogía con el mundo real, el código resulta más fácil de entender. Cuando no la existe, surgen dudas. Dudas por otro lado razonables, porque la terminología Java podemos decir que es un poco confusa: se nos habla de objetos y cuando pensamos en objetos tendemos a pensar en lo que en el día a día se considera un objeto: algo que se puede tocar. Y por derivación suponemos que un objeto en programación orientada a objetos será la representación de un objeto de la vida real. Pero esto no es así: un objeto en Java puede representar un objeto de la vida real, pero también puede ser algo abstracto e intangible. Tendremos que considerarlo un simple espacio de memoria del ordenador con capacidad para realizar procesos. Y esto resulta un poco más difícil de asimilar. La mejor forma de hacerlo es, a nuestro juicio, ir paso a paso creando pequeños programas y descubriendo las posibilidades de Java. Entender bien el paradigma de programación orientada a objetos requiere tiempo.

La definición de objeto que habíamos dado en epígrafes anteriores sigue siendo válida:

Objeto: entidad existente en la memoria del ordenador que tiene unas propiedades (atributos o datos sobre sí mismo almacenados por el objeto) y unas operaciones disponibles específicas (métodos).

La cuestión es que **en algunos casos un objeto no podemos asociarlo a nada en el mundo real**. Además, un objeto en algunas circunstancias puede carecer de atributos, carecer de constructor, o carecer de métodos. Un objeto de tipo Exponenciador según el ejemplo que hemos visto, se correspondería con el caso de objeto sin atributos y sin constructor. En realidad hemos preferido incluir un constructor vacío indicando que no teníamos nada que declarar. Si no lo hubiésemos hecho así,

quién leyera el código podría tener la duda de si nos hemos “olvidado” de incluir el constructor. Este objeto es capaz de ejecutar dos métodos que reciben parámetros y devuelven un resultado, pero internamente no almacena ninguna información como podía ser la matrícula o el distrito de los objetos Taxi.

La clase sigue definiendo un tipo, pero esto es quizás ahora menos relevante. En este caso preferimos verla como un paquete de código que nos permite realizar procesos.

EJERCICIO

Define una clase denominada multiplicadorDieces con un constructor vacío y que contenga un método denominado multiplicarPorDieces que reciba dos parámetros: el primero un número de tipo double y el segundo un número de tipo entero. El método debe devolver el resultado de multiplicar por 10 elevado al segundo número el primer número. Ejemplo: multiplicarPorDieces (2.55, 2) devuelve $2.55 \times 10^2 = 255$. multiplicarPorDieces (3, 5) devuelve $3 \times 10^5 = 300000$. MultiplicarPorDieces (-0.0563, 3) devuelve $-0.0563 \times 10^3 = -56.3$. Crea un objeto y comprueba que el método opera correctamente. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00645B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ ES Y PARA QUÉ SIRVE EL API DE JAVA?

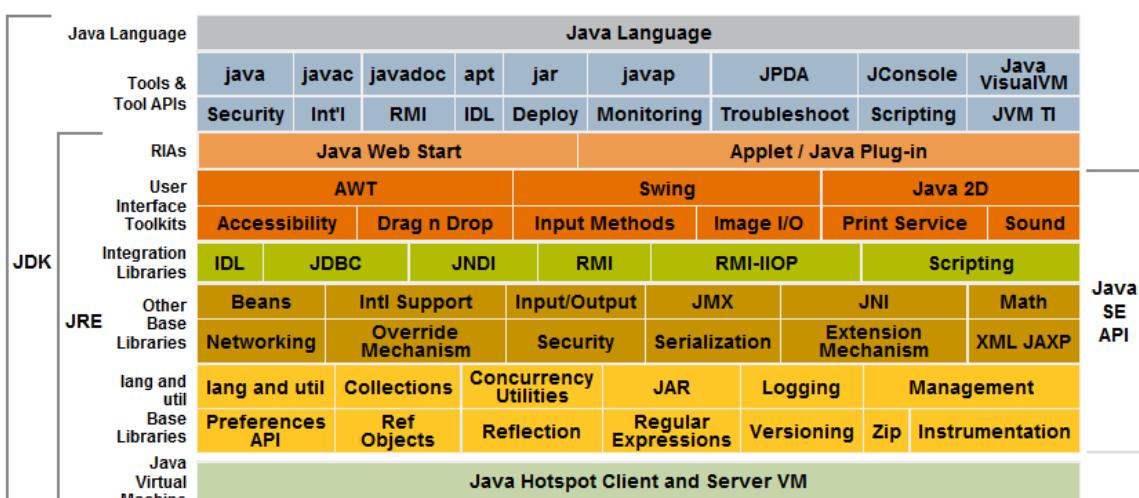
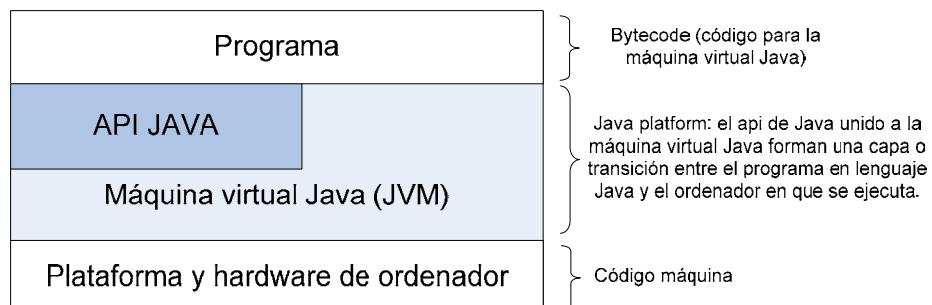
Hasta ahora hemos visto ejemplos donde utilizábamos la clase `System` (por ejemplo en la invocación `System.out.println`) o la clase `String` (que es la clase gracias a la que podemos usar los objetos `String`). ¿De dónde salen estas clases si nosotros no las hemos programado como la clase `Taxi` o la clase `Exponenciador`?



La respuesta está en que al instalar Java (el paquete JDK) en nuestro ordenador, además del compilador y la máquina virtual de Java se instalan bastantes más elementos. Entre ellos, una cantidad muy importante de clases que ofrece la multinacional desarrolladora de Java y que están a disposición de todos los programadores listas para ser usadas. Estas clases junto a otros elementos forman lo que se denomina API (Application Programming Interface) de Java.

La mayoría de los lenguajes orientados a objetos ofrecen a los programadores bibliotecas de clases que facilitan el trabajo con el lenguaje.

Los siguientes esquemas, parte de la documentación de Java, nos dan una idea de cómo funciona el sistema Java y de qué se instala cuando instalamos Java (el paquete JDK) en nuestro ordenador.



Este esquema no nos interesa analizarlo en profundidad. Lo único que queremos mostrar es que **cuando instalamos Java en nuestro ordenador instalamos múltiples herramientas**, entre ellas una serie de “librerías” (paquetes) a cuyo conjunto solemos referirnos como “biblioteca estándar de Java”. Las librerías contienen código Java listo para ser usado por nosotros. Ese es el motivo de que podamos usar clases como System o String sin necesidad de programarlas.

¿Dónde se encuentra el código de estas librerías? En los archivos que se instalan en nuestro ordenador cuando instalamos Java.

¿Podemos acceder al código de estas librerías? La respuesta es que no. Las biblioteca estándar de Java se facilita como código cerrado, es decir, como código máquina. No podemos acceder al código fuente. Esto tiene su lógica porque si cada persona accediera al código fuente de los elementos esenciales de Java y los modificara, no habría compatibilidad ni homogeneidad en la forma de escribir programas Java.

¿Para qué nos sirve la biblioteca si no podemos acceder a su código fuente? Aunque no podemos acceder al código fuente, sí podemos crear objetos de los tipos definidos en la librería e invocar sus métodos. Para ello lo único que hemos de hacer es conocer las clases y la signatura de los métodos, y esto lo tenemos disponible en la documentación de Java accesible para todos los programadores a través de internet o cds de libros y revistas especializadas. ¿Te acuerdas cuando hablamos de la definición de signatura e interfaz? Esto es ahora plenamente aplicable con el API de Java: tenemos disponible información de las clases, de qué hacen y cómo podemos trabajar con ellas, aunque no dispongamos del código fuente.

Próxima entrega: CU00646B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

ORGANIZACIÓN Y FORMA DE NOMBRAR LAS LIBRERÍAS DEL API DE JAVA

La biblioteca estándar de Java está compuesta por cientos de clases como *System*, *String*, *Scanner*, *ArrayList*, *HashMap*, etc. que nos permiten hacer casi cualquier cosa. Imagínate que quieras crear una lista de países donde ir añadiendo nombres de países y en un momento dado ordenarlos por orden alfabético.



O supón que quieras tener una lista de países relacionados con su continente (p. ej. México <-> América, España <-> Europa, Argentina <-> América, etc.). Para tener una primera idea de si el API de Java contendrá clases que nos puedan servir de ayuda, nos podemos hacer la pregunta: ¿habrán tenido muchos programadores necesidad de herramientas de este tipo? La respuesta es que obviamente sí. Ordenar, tener clasificado, hacer operaciones matemáticas, hacer búsquedas de texto, pedir datos al usuario y muchos más procesos son cuestiones que se repiten con frecuencia en programación, y por tanto se encontrarán resueltas en el API de Java. Por supuesto que podemos crear algoritmos propios para ordenar listas, pero lo más rápido y eficiente en general será usar las herramientas del API disponibles porque están desarrolladas por profesionales y han sido depuradas y optimizadas a lo largo de los años y versiones del lenguaje.

Saber usar la biblioteca y elegir las clases adecuadas es esencial para crear programas de forma rápida y eficiente. Es imposible conocer todas las clases y sus detalles (constructores, campos, métodos, etc.), ni siquiera sus nombres, pero gracias a internet lo tenemos todo “al alcance de la mano”.

Para programar en Java tendremos que recurrir continuamente a consultar la documentación del API de Java. Esta documentación está disponible en cds de libros y revistas especializadas o en internet tecleando en un buscador como yahoo, google o bing el texto “api java 8” (o “api java 6”, “api java 10” etc.) según la versión que estés utilizando. La documentación del API de Java en general es correcta y completa. Sin embargo, en casos excepcionales puede estar incompleta o contener erratas.

Cuando tengamos experiencia como programadores Java, posiblemente dispongamos de clases desarrolladas por nosotros mismos que utilicemos en distintos proyectos. En empresas grandes, es frecuente disponer de clases desarrolladas por compañeros de la empresa que usaremos de forma parecida a como se usa el API de Java: conociendo su interfaz pero no su implementación. Trabajar con una clase sin ver su código fuente requiere que exista una buena documentación que nos sirva de guía. Hablaremos de la documentación de las clases y proyectos en Java un poco más adelante. De momento, vamos a aprender a usar la documentación del API de Java.

En primer lugar, debemos tener una idea de cómo se organizan las clases del API. Esta organización es en forma de árbol jerárquico, como se ve en la figura “Esquema orientativo de la organización de librerías en el API de Java”. Esta figura trata de mostrar la organización del API de Java, pero no recoge todos los paquetes ni clases existentes que son muchos más y no cabrían ni en una ni en varias hojas.

Los nombres de las librerías responden a este esquema jerárquico y se basan en la notación de punto. Por ejemplo el nombre completo para la clase ArrayList sería `java.util.ArrayList`. Se permite el uso de * para nombrar a un conjunto de clases. Por ejemplo `java.util.*` hace referencia al conjunto de clases dentro del paquete `java.util`, donde tenemos `ArrayList`, `LinkedList` y otras clases.

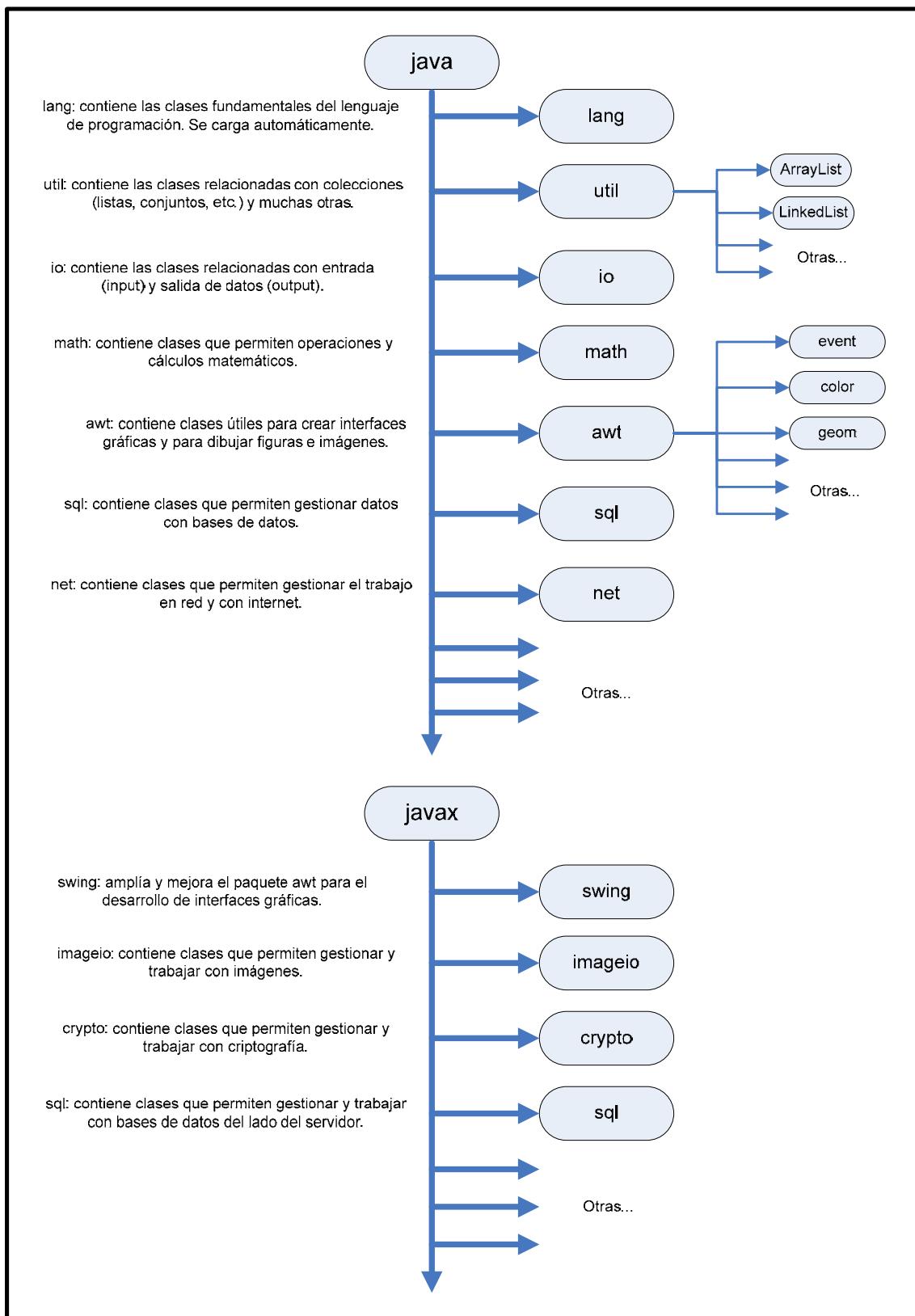
Para utilizar las librerías del API, existen dos situaciones:

- a) Hay **librerías o clases que se usan siempre** pues constituyen elementos fundamentales del lenguaje Java como la clase `String`. Esta clase, perteneciente al paquete `java.lang`, se puede utilizar directamente en cualquier programa Java ya que se carga automáticamente.
- b) Hay **librerías o clases que no siempre se usan**. Para usarlas dentro de nuestro código hemos de indicar que requerimos su carga mediante una sentencia `import` incluida en cabecera de clase. Por ejemplo `import java.util.ArrayList;` es una sentencia que incluida en cabecera de una clase nos permite usar la clase `ArrayList` del API de Java. Escribir `import java.util.*;` nos permitiría cargar todas las clases del paquete `java.util`. Algunos paquetes tienen decenas o cientos de clases. Por ello nosotros preferiremos en general especificar las clases antes que usar asteriscos ya que evita la carga en memoria de clases que no vamos a usar. Una clase importada se puede usar de la misma manera que si fuera una clase generada por nosotros: podemos crear objetos de esa clase y llamar a métodos para operar sobre esos objetos. Además cada clase tendrá uno o varios constructores.

Las librerías podemos decir que se organizan en ramas como si fueran las ramas de un árbol. Vamos a fijarnos en dos grandes ramas: la rama “`java`” y la rama “`javax`”. La rama `java` parte de los orígenes de Java, mientras que la rama `javax` es más moderna. iremos conociendo poco a poco tanto la una como la otra.

Encontrar un listado de librerías o clases más usadas es una tarea casi imposible. **Cada programador, dependiendo de su actividad, utiliza ciertas librerías** que posiblemente no usen otros programadores. Los programadores más centrados en programación de escritorio usarán clases diferentes a las que usan programadores web o de gestión de bases de datos. Las clases y las librerías básicas deberás ir conociéndolas mediante cursos o textos de formación básica en Java. Las clases y librerías más avanzadas deberás utilizarlas y estudiarlas a medida que te vayan siendo necesarias para el desarrollo de aplicaciones, ya que su estudio completo es prácticamente imposible. Podemos citar clases de uso amplio. En el paquete `java.io`: clases `File`, `Filewriter`, `Filereader`, etc. En el paquete `java.lang`: clases `System`, `String`, `Thread`, etc. En el paquete `java.security`: clases que permiten implementar encriptación y seguridad. En el paquete paquete `java.util`: clases `ArrayList`, `LinkedList`, `HashMap`, `HashSet`, `TreeSet`, `Date`, `Calendar`, `StringTokenizer`, `Random`, etc. En los paquetes `java.awt` y `javax.swing` una biblioteca gráfica: desarrollo de interfaces gráficas de usuario con ventanas, botones, etc.

Insistimos en una idea: no trates de memorizar la organización detallada del API de Java ni un listado de clases más usadas porque esto tiene poco sentido. Lo importante es que conozcas la forma de organización, cómo se estructuran y utilizan las clases y que aprendas a buscar información para encontrarla rápidamente cuando te sea necesaria.



Esquema orientativo de la organización de librerías en el API de Java

Próxima entrega: CU00647B**Acceso al curso completo** en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

IMPORTAR Y USAR CLASES DEL API DE JAVA. EJEMPLO CON LA CLASE MATH.

En el API de Java no solo existen clases. Hay bastantes más cosas como clases abstractas o interfaces, de lo que hablaremos más adelante. De momento vamos a centrarnos en las clases y para ello hemos de prestar atención a aquello que nos aparece en el encabezado de la documentación que consultemos. Realiza en internet una búsqueda con el texto “math api java X” donde X será la versión de java que estemos usando.



math api java 6

Aproximadamente 30.500.000 resultados (0,19 segundos) Google.com in English Búsqueda avanzada

- ▶ [Math \(Java Platform SE 6\)](#)   - [Traducir esta página]
download.oracle.com/javase/6/docs/api/java/lang/Math.html - En caché
The class **Math** contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric ...
- ▶ [BigDecimal \(Java Platform SE 6\)](#)   - [Traducir esta página]
download.oracle.com/javase/6/.../api/java/math/BigDecimal.h... - En caché
java.lang.Object extended by java.lang.Number extended by **java.math** ...
- ▶ [BigInteger \(Java Platform SE 6\)](#)   - [Traducir esta página]
download.oracle.com/javase/6/.../api/java/math/BigInteger.h... - En caché
BigInteger provides analogues to all of Java's primitive integer operators ...
- ▶ [java.math \(Java Platform SE 6\)](#)   - [Traducir esta página]
download.oracle.com/javase/6/.../api/java/math/package-su... - En caché
Package **java.math**. Provides classes for performing arbitrary-precision ...
- [Mostrar más resultados de oracle.com](#)
- ▶ [Java: Random numbers - API](#)   - [Traducir esta página]
leepoint.net/notes-java/algorithms/random/random-api.html - En caché

Nos pueden aparecer diferentes resultados ya que **Math** en Java puede hacer referencia a distintas cosas, por ejemplo al paquete `java.math` o a la clase `Math`. Buscaremos el correspondiente a la clase `Math` y accedemos a él.

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: NESTED | [FIELD](#) | [CONSTR](#) | [METHOD](#)

`java.lang`
Class Math

`java.lang.Object`
└ `java.lang.Math`

```
public final class Math
extends Object
```

The class **Math** contains methods for performing basic numeric operations such

Fíjate que en la cabecera de la documentación se indica “Class Math”. Si nos indicara otra cosa como Interface Math, Package Math, o cualquier otro texto, no nos encontraríamos frente a la documentación de la clase. Fíjate también en la ruta de la clase dentro del API de Java que aparece expresada como java.lang.Math. Es decir, la clase Math pertenece al paquete java.lang del API de Java. Vamos a buscar ahora el apartado “Method Summary” y dentro de éste el apartado *pow*.

static double	pow(double a, double b)
	Returns the value of the first argument raised to the power of the second argument.

Aquí nos vamos a fijar en la columna izquierda, que nos indica el tipo devuelto y que resulta ser *double*. En la columna derecha se indica la firma del método: el método se llama *pow* y requiere dos valores numéricos de tipo *double*. El texto explicativo nos indica que el método devuelve el valor del primer parámetro elevado a la potencia indicada por el segundo parámetro. Escribe e intenta compilar el siguiente código (no nos va a ser posible el compilado):

```
import java.lang.Math; //Importamos la clase Math de la biblioteca del API Java

//Clase que permite elevar un número m a otro número n y obtener un resultado
public class ExponenciadorApiJava {

    //Constructor
    public ExponenciadorApiJava () { //Nada que declarar
    }

    public int potenciaApiJava (int m, int n) {
        double a = new Math();
        return a.pow (m, n);
    } //Cierre del método
} //Cierre de la clase
```

En la primera línea de código, antes de la declaración de la clase, hemos incluido una sentencia import y una ruta del API de Java. Con este tipo de sentencias lo que hacemos es indicarle al compilador que para la ejecución del programa cargue en memoria el código contenido en la ruta indicada. Esto es lo que nos permite usar clases del API de Java.

Al tratar de compilar nos salta un error de tipo “Math() has private access in java.lang.Math”. Lo cual nos quiere decir que el constructor de la clase Math tiene acceso privado en la clase Math. Hasta ahora hemos visto cómo los constructores los declarábamos siempre de acceso público, pero **en ciertas ocasiones se declaran constructores privados**, lo que significa que no están accesibles y no podemos hacer invocaciones usando la sentencia *new*. Si revisas la documentación del API de la clase Math, nos encontramos con dos tablas: *Field Summary* (campos o atributos de la clase) y *Method Summary* (métodos de la clase). En muchas otras clases nos encontramos con otra tabla denominada *Constructor Summary* que nos indica cómo crear objetos de la clase. Sin embargo, en la clase Math esa tabla falta. El motivo para ello es que dentro del API de Java algunas clases tienen un comportamiento especial, y la

clase Math es una de ellas. Es una clase en la que la gestión de objetos queda a cargo de Java y a nosotros únicamente se nos permite invocar métodos. La sintaxis que define Java en estos casos es del tipo:

nombreDeLaClase.nombreDelMétodo (parámetros requeridos);

Por ejemplo: `Math.pow (m, n);`

¿Contradice esto el principio de que los métodos se invocan sobre objetos? Podemos responder que sí y que no. Sí porque efectivamente estamos invocando el método usando el nombre de una clase. Y no porque Java, en segundo plano, está utilizando un objeto que crea automáticamente para realizar la gestión de esta invocación. Por tanto la clase Math define un tipo, pero nosotros no vamos a poder crear objetos de ese tipo, de ello se encargará Java. Nosotros veremos la clase Math principalmente como un paquete de código que nos permitirá realizar operaciones matemáticas. Que la clase Math defina un tipo será algo secundario para nosotros. Hablaremos de estos métodos, denominados estáticos, más adelante.

Realicemos un nuevo intento de compilación teniendo en cuenta la forma de invocación para la clase Math:

```
import java.lang.Math;
//Clase que permite elevar un número m a otro número n y obtener un resultado
public class ExponenciadorApiJava {
    public ExponenciadorApiJava () {} //Nada que declarar

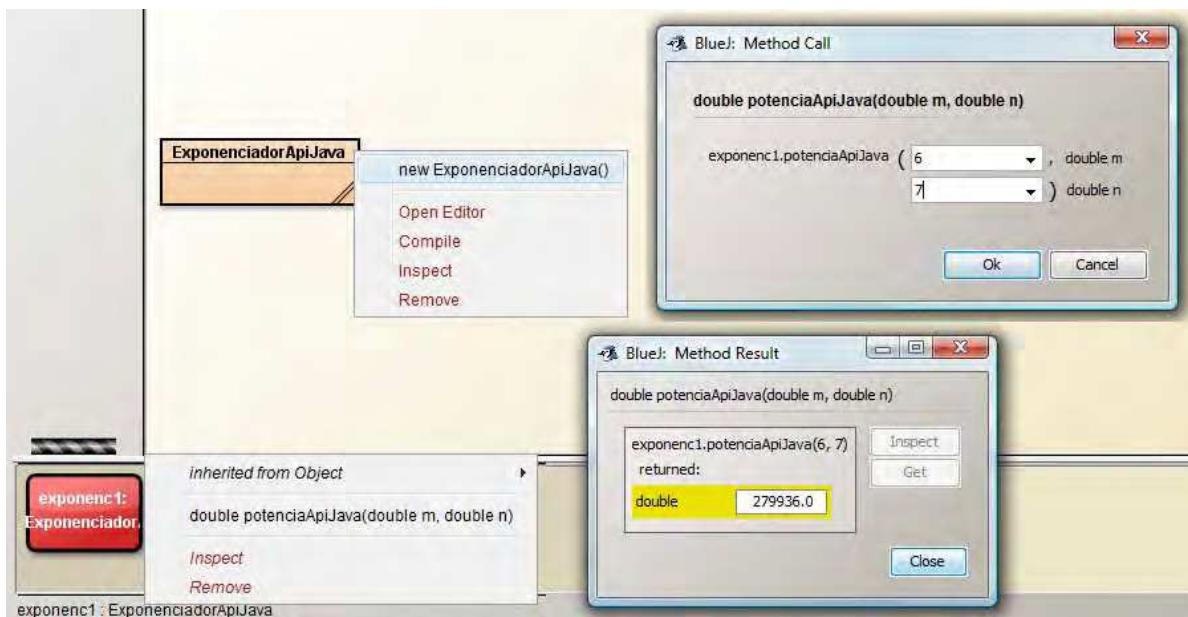
    public int potenciaApiJava (int m, int n) { return Math.pow (m, n); } //Cierre del método
} //Cierre de la clase
```

Volvemos a obtener un error de compilación del tipo “possible loss of precision found: double required: int”. En este caso el compilador nos indica que el método `pow` devuelve un valor de tipo `double` cuando el método en su firma indica que se va a devolver un valor de tipo `int` y eso puede ser problemático. El método `potenciaApiJava` lo tenemos definido ahora mismo como un método que devuelve un entero y recibe como parámetros dos enteros. Sin embargo, el método `pow` de la clase Math devuelve un `double` y espera recibir como parámetros dos valores de tipo `double`. Hay varias maneras de resolver esto. Nosotros vamos a optar ahora por ceñirnos a lo que nos indica el API de Java y vamos a cambiar la firma de nuestro método para que devuelva y reciba valores de los tipos esperados. Intentaremos compilar teniendo en cuenta lo dicho anteriormente:

```
import java.lang.Math;
//Clase que permite elevar un número m a otro número n y obtener un resultado
public class ExponenciadorApiJava {
    public ExponenciadorApiJava () {} //Nada que declarar

    public double potenciaApiJava (double m, double n) {
        return Math.pow (m, n);
    } //Cierre del método
} //Cierre de la clase
```

Crea un objeto de tipo ExponenciadorApiJava e invoca su método para obtener las potencias de 2^3 y de 6^7 . Los resultados deben ser los mismos que los obtenidos en los ejemplos anteriores, aunque ahora se nos mostrará un resultado terminado en .0 para indicar que tiene precisión decimal.



Prueba ahora a eliminar la sentencia *import* de la primera línea y a compilar. La compilación es posible. ¿Por qué? Esto se debe a lo que comentamos relativo a que **determinadas clases o paquetes se cargan automáticamente** mientras que otros no se cargan a no ser que se indique específicamente. El paquete `java.lang` es un paquete que se carga automáticamente. Por ello podemos hacer uso de todas sus clases, como `String` o `Math`, sin necesidad de importarlo. Si escribimos la sentencia de importación no habrá mensaje de error, pero tampoco será útil ya que estamos redundando al repetir algo que hace Java automáticamente.

EJERCICIO

Una operación frecuente con valores numéricos puede ser obtener su valor absoluto, es decir, su valor sin signo. Por ejemplo el valor absoluto de -3.22 es 3.22 (hemos eliminado el signo negativo) y el valor absoluto de 7.15 es 7.15 (al ser este número positivo coincide con su valor absoluto). Otra operación frecuente es el cálculo de la raíz cuadrada de un número. Consulta la documentación de la clase `Math` del api Java para comprobar qué métodos permiten realizar estas tareas. Crea una clase denominada `miniCalculadoraEjemplo` que tenga dos métodos (basados en el uso de métodos de la clase `Math`): un método `valorAbsoluto` que recibe un número de tipo `double` y devuelva su valor absoluto, y otro método `raizCuadrada` que reciba un número de tipo `double` y devuelva su raíz cuadrada. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00648B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ ES UNA INTERFACE DE CLASE JAVA? CONCEPTO. EJEMPLO CON LA CLASE STRING.

Supongamos que queremos determinar si una cadena (String) comienza de una manera determinada, por ejemplo si empieza por ‘caza’. Esta condición la cumplirían cadenas como “cazador”, “cazaculebras” y “caza prohibida”. Nosotros podríamos desarrollar un método que nos permita realizar esta determinación usando código propio.



Pero dado que queremos actuar sobre un objeto del API de Java (un String) y dado que queremos hacer algo que con toda seguridad es un problema que se le presenta con frecuencia a muchos programadores, lo lógico es consultar la documentación de la clase. Posiblemente ese método se encuentre disponible dentro de los métodos de la clase en el API de Java y nos podamos ahorrar tiempo y código si lo utilizamos.

Si en un buscador de internet introducimos el texto “api java X” donde X es la versión de java que estemos usando (por ejemplo “api java 9”), podemos acceder a una vista resumen de los paquetes y clases del API de Java. Otra forma de acceder es a través de BlueJ. Vamos al menú Help → Java Class Libraries (biblioteca de clases Java), y se nos abre el navegador en la página del API de Java de la versión de Java que esté usando BlueJ.

Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.

La información se suele distribuir en marcos. En uno de ellos (1) el detalle de la clase o paquete que tengamos seleccionado. En otro marco (2), el listado de paquetes (librerías). Y en otro marco (3), el listado de clases (“All Classes”, todas las clases disponibles). Para buscar la documentación de una clase buscamos la clase en el listado de clases. Vamos a buscar la clase String y pulsamos sobre ella.

En la ventana de detalle veremos que la clase String tiene una extensa documentación. Prueba a buscar la clase ArrayList y échale un vistazo a su documentación. Verás que la documentación es un poco menos extensa, pero la estructura de la documentación es similar para todas las clases y suele comprender:

- 1.- El nombre de la clase y una descripción general.
- 2.- Lista breve de campos (atributos) de la clase (Fields).
- 3.- Lista breve de constructores de la clase.
- 4.- Lista breve de métodos de la clase.
- 5.- Lista detallada de los campos.
- 6.- Lista detallada de los constructores.
- 7.- Lista detallada de los métodos.

Toda esta información que describe qué hace la clase y cómo usarla (sin mostrar el código fuente o implementación) se denomina interfaz o interface de la clase. El código de implementación de la clase queda oculto (principio de ocultamiento de la información) y como programadores no vamos a tener acceso ni vamos a necesitar tener acceso a él.

La interfaz de clase nos muestra todos los constructores y métodos que se hayan definido como *public* en la clase. Por el contrario, no se van a mostrar aquellos constructores o métodos que se hayan definido como *private*. El motivo para ello es que los constructores o métodos *private* se consideran código auxiliar para su uso exclusivo dentro de la clase al que no se debe tener acceso desde fuera de ella.

El conjunto de signaturas de métodos y constructores públicos de una clase constituyen su interfaz o interface. En esencia, la interface es una abstracción que consiste en que conocemos la signatura de los métodos (qué hacen) pero no su implementación (cómo lo hacen). Muchas veces se hace referencia a la implementación como “parte privada de una clase” para distinguirla de la parte pública (interfaz). Esta forma de trabajar se dice que hace uso del principio de ocultamiento de la información y se ha demostrado que es beneficiosa para una buena programación.

Veamos en síntesis (no la veremos de forma completa porque resulta realmente extensa) los contenidos que ofrece la documentación de la clase String. Ten en cuenta que puede haber algunos cambios según la versión de Java que se emplee.

Nombre de la clase y descripción general

java.lang

Class String

```
java.lang.Object
  └─ java.lang.String
```

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The string class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class. Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

Continúa comentarios, ejemplos y cuestiones relevantes de la clase...

Since:

JDK1.0

See Also:

[Object.toString\(\)](#), [StringBuffer](#), [StringBuilder](#), [Charset](#), [Serialized Form](#)

Lista breve de campos (Field Summary)

Field Summary

static Comparator<String>	CASE_INSENSITIVE_ORDER
---------------------------	-------------------------------

A Comparator that orders String objects as by compareToIgnoreCase.

Lista breve de constructores (Constructor Summary)

Constructor Summary

[String\(\)](#)

Initializes a newly created String object so that it represents an empty character sequence.

[String\(char\[\] value\)](#)

Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

Continúa constructores de la clase...

Lista breve de métodos

Method Summary

String	concat(String str) Concatenates the specified string to the end of this string.
boolean	contains(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
int	length() Returns the length of this string.
boolean	startsWith(String prefix) Tests if this string starts with the specified prefix.
boolean	startsWith(String prefix, int toffset) Tests if the substring of this string beginning at the specified index starts with the specified prefix.
String	substring(int beginIndex) Returns a new string that is a substring of this string.

String	substring (int beginIndex, int endIndex) Returns a new string that is a substring of this string.
Continúa métodos de la clase...	

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Detalle de campos (Field Detail), constructores (Constructor Detail) y métodos (Method Detail)

En la parte inferior de la documentación se encuentra información de detalle (más extensa) sobre los elementos que se muestran en las listas breves o sumarios. Podemos acceder a ellos pulsando sobre el elemento correspondiente de la lista breve o bien recorriendo completamente la documentación. En distintas partes (encabezado, final) de la documentación nos aparece la información de a qué versión de Java corresponde esa documentación y qué es lo que se documenta:



The screenshot shows a portion of the Java SE Developer Documentation. At the top, there's a navigation bar with links: Overview, Package, Class Tree, Deprecated, Index, Help. Below the navigation bar, there's some small text and a copyright notice. On the right side, there's a large oval-shaped callout containing the text "Java™ Platform Standard Ed. 6".

Hemos señalado dos cosas a las que debemos de prestar atención: a qué corresponde la documentación que estamos consultando. Corresponde a una clase y a Java Platform Standard Ed. X (Java SE X) donde la X es la versión de Java. Si no realizamos bien las búsquedas puede ocurrir que estemos consultando la documentación de algo que no sea una clase, o de una distribución de Java que no sea aquella con la que estemos trabajando. A nivel profesional, es posible que nos veamos en la tesitura de tener que trabajar en algunos proyectos con Java SE 6 y en otros con Java SE 12 o posteriores. Hemos de prestar la atención necesaria para diferenciarlos y para no confundirnos.

Hemos dicho que dada una interfaz de una clase no vamos a necesitar ver el código de implementación de la clase. ¿Es esto siempre cierto? Digamos que sí siempre que la interface esté bien redactada y documentada, como es el caso de la biblioteca estándar Java. Si estamos trabajando en una empresa y nos facilitan una interfaz de clase desarrollada por otras personas y esta interfaz es pobre, incompleta o mal redactada, es posible que nos veamos obligados a solicitar consultar el código fuente para entender la clase. En proyectos grandes, es frecuente que sea necesario definir las interfaces de las clases antes incluso de que exista el código de implementación. El sentido que tiene esto es que podamos desarrollar código incluyendo las formas de invocación previstas en la signatura de las clases, independientemente de que el código de esas clases tarde en desarrollarse o, una vez desarrollado, vaya sufriendo cambios o mejoras. **La idea que subyace al concepto de interfaz es la abstracción:** saber qué hace y cómo usar una clase, pero “olvidarnos” de su implementación (por implementación entendemos el código completo que define una clase). La distinción entre interfaz e implementación es un concepto clave en programación orientada a objetos.

Próxima entrega: CU00649B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

ESTUDIANDO EL CONCEPTO DE MÉTODO JAVA. EL MÉTODO SUBSTRING DE LA CLASE STRING

Vamos a centrar nuestra atención en la documentación del método substring de la clase String. La primera cuestión en que nos fijamos es que existen dos formas de invocar el método con el mismo nombre pero con distintos parámetros: se trata de un método sobrecargado.



En el resumen del método comprobamos que el método devuelve un tipo String y nos pide como parámetros uno o dos valores enteros.

<u>String</u>	substring(int beginIndex) Returns a new string that is a substring of this string.
<u>String</u>	substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.

Pero no queda muy claro qué son esos valores enteros que nos pide. Para aclararlo pinchamos sobre el nombre del método y vemos la documentación de detalle. Con la documentación de detalle ya queda claro cómo podemos usar el método. Vamos a centrarnos en la forma de uso empleando dos parámetros. Podemos extraer una fracción de una cadena usando esta sintaxis:

```
fraccionDeString = nombreDelString.substring (carácter Inicial Incluido, carácter Final Excluido)
```

Por ejemplo: `fraccion1 = miCadena.substring (0, 4);` extrae los cuatro primeros caracteres de miCadena, que podemos nombrar como 0, 1, 2, 3 (el número de caracteres es cuatro). El carácter final indicado en la llamada al método, el número 4, queda excluido del substring.

Otro ejemplo: `return nombre.substring (0, 4) + id.substring (0, 3);` devuelve una cadena donde los 4 primeros caracteres se han extraído del String denominado *nombre* y los tres siguientes del String denominado *id*. Tanto *nombre* como *id* tienen que ser obligatoriamente tipo String, ya que si alguno de ellos no lo fuera no sería posible aplicarles el método substring.

Otros ejemplos: “hamburger”.substring (4, 8); devuelve “urge”.

“smiles”.substring (1, 5) devuelve “mile”.

“coco”.substring (2, 2) devuelve “” (cadena vacía).

“coco”.substring (7, 3) devuelve un error ya que no existe un carácter número 7 dentro de la cadena.

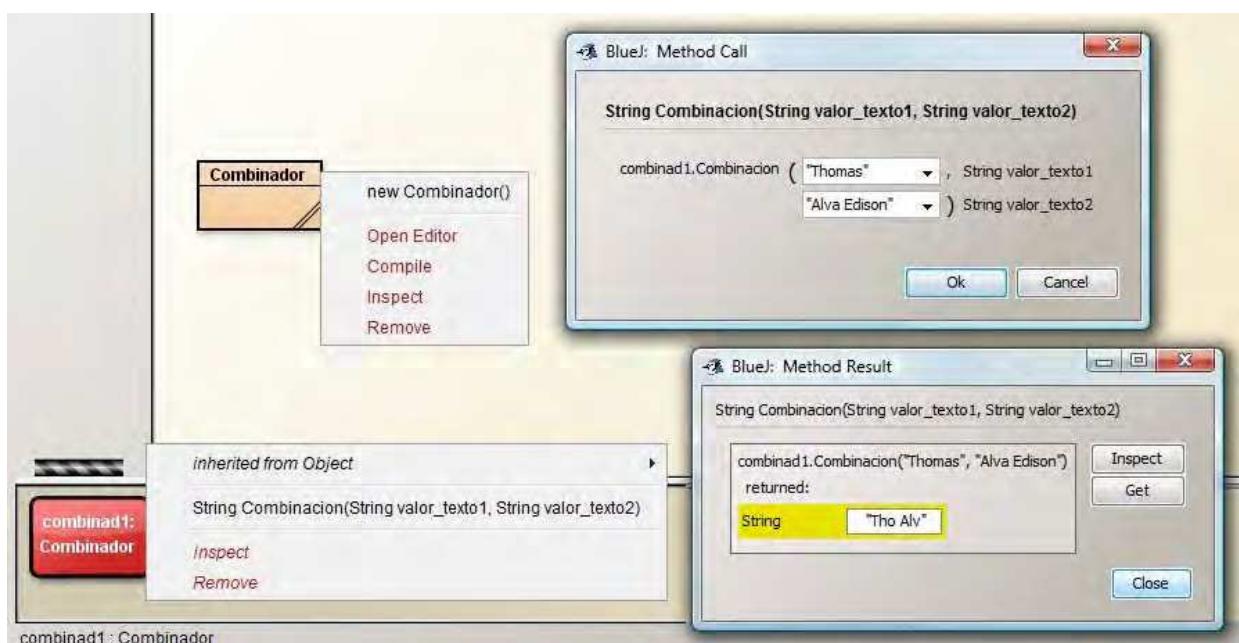
Un fragmento de código usando substring puede ser este:

```
//Combina las tres primeras letras de dos textos recibidos como parámetro en una sola cadena separada por un espacio
public class Combinador {
    //Campos de la clase
    private String texto1;
    private String texto2;

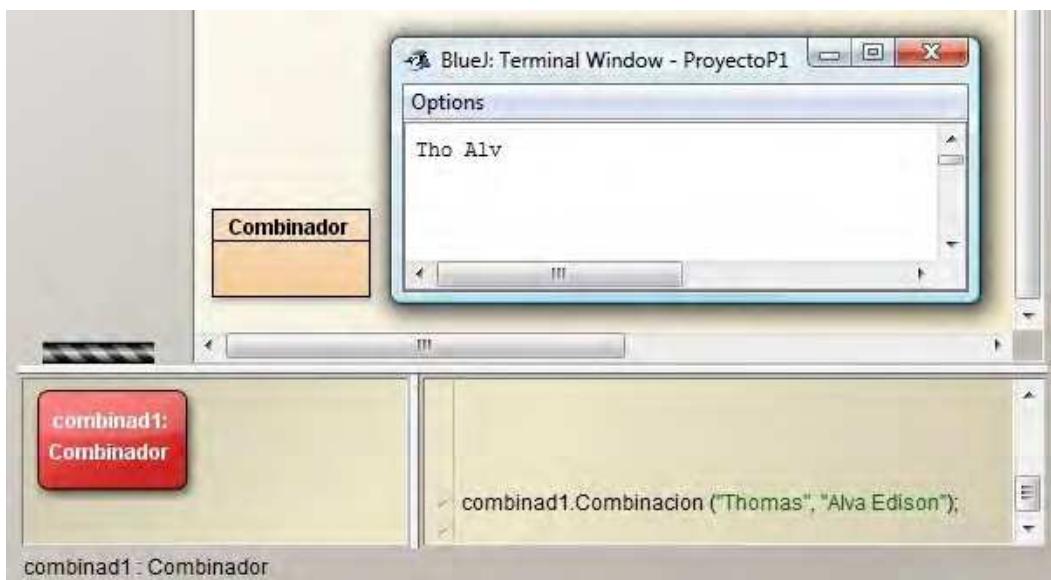
    //Constructor de la clase
    public Combinador () {
        texto1 = "";
        texto2 = "";
    } //Cierre del constructor

    //Método que combina las tres primeras letras de cada cadena
    String combinacion (String valor_texto1, String valor_texto2) {
        String combinacion = valor_texto1.substring (0,3) + " " + valor_texto2.substring (0,3);
        return combinacion; //combinacion es tanto el nombre del método como el de una variable local
    } //Cierre del método
} //Cierre de la clase
```

Crea un objeto de tipo Combinador y ejecuta el método *combinacion* introduciendo dos cadenas como “Thomas” y “Alva Edison” y comprueba sus resultados.



Para ir conociendo BlueJ, vamos a fijarnos ahora en la pequeña ventana que aparece en la parte inferior derecha de la ventana principal (si no está activa, actívala en el menú View → Show Codepad). Esta ventana, denominada Codepad, nos permite escribir código utilizando los objetos que tenemos en el banco de objetos. Si el objeto que has creado se llama *combinad1*, puedes escribir el siguiente código: *combinad1.combinacion (“Thomas”, “Alva Edison”);* y pulsa enter. Para comprobar que el método se está ejecutando introduce en el código una instrucción para que se muestre el resultado en la ventana de consola.



Como resumen de lo visto en este apartado podemos decir lo siguiente. String es una clase en Java y substring un método tipo función de la clase que devuelve una cadena de texto. La cadena devuelta es una fracción de la cadena sobre la que se invoca el método. El método substring está sobrecargado porque existe más de una forma de llamarlo. La signatura del método substring podemos consultarla en la documentación del API de Java, lo que nos resulta suficiente para utilizar el método sin necesidad de conocer el código de implementación del mismo (esta información nos queda oculta).

Si hacemos una llamada al método substring pidiendo que se extraiga un número de caracteres superior al número disponible en la cadena se produce un error de tipo "java.lang.StringIndexOutOfBoundsException. String index out of range". Podríamos evitar este tipo de errores si controláramos cuál es la longitud o número de caracteres de las cadenas con las que trabajamos. Para ello podemos recurrir una vez más al API de Java, en concreto al método *length* de los objetos tipo String.

EJERCICIO

Crea una clase (ponle el nombre que quieras) que carezca de atributos y cuyo constructor esté vacío. En dicha clase debe existir un método tipo función que reciba 3 cadenas de texto y devuelva la cadena de texto combinación de: las dos primeras letras de la primera cadena, seguidas por un espacio en blanco, seguidas de las cuatro primeras letras de la segunda cadena, seguidas de un guión medio y seguido de las 6 primeras letras de la tercera cadena. Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00650B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

USAR MÉTODOS PARA EVITAR ERRORES. EJEMPLO MÉTODO LENGTH DE LA CLASE STRING

Vamos a usar el método *length* de la clase String para tratar de evitar que nos salten errores al emplear el método *substring*. Lo primero que haremos será consultar la documentación de la clase, la firma del método, y, si es necesario, su detalle. La firma del método es:



<u>int</u>	<u>length()</u>
------------	-----------------

Devuelve el número de caracteres del String.

El tipo devuelto por el método es int (un entero) y el método no requiere parámetros para ser ejecutado. El método no está sobrecargado, ya que hay una única manera de invocarlo. Un ejemplo de invocación puede ser *NumCaracteres = micadena.length();*. Recordar que **siempre que invoquemos un método hemos de incluir los paréntesis**, independientemente de que sea necesario pasar parámetros o no.

Valiéndonos de este método, intenta escribir tu propio código para hacer lo siguiente:

- Reescribir la clase Combinador que usamos anteriormente de forma que el método *combinacion* devuelva “No aporta cadenas válidas” si se le pasa como parámetro alguna cadena que contenga menos de tres caracteres.
- Escribir un nuevo método de la clase, al que podemos denominar *combinadoSiempre* que en caso de que se pasen cadenas con menos de tres caracteres, las combine de todas formas en base a los caracteres disponibles, sea el número que sea.

El código podría ser el siguiente:

```
// Definición de una clase de ejemplo con uso del método length sobre objetos String aprenderaprogramar.com
// Combina las tres primeras letras de dos textos introducidos por el usuario en una sola cadena separada por un espacio
public class Combinador {

    private String texto1;
    private String texto2;
    public Combinador () { texto1 = ""; texto2 = "" }

    //Método que combina las tres primeras letras de cada cadena
    String Combinacion (String valor_texto1, String valor_texto2) {
        if (valor_texto1.length() >= 3 && valor_texto2.length() >= 3) { //Comprobación
            String combinacion = valor_texto1.substring (0, 3) + " " + valor_texto2.substring (0 ,3);
            return combinacion;
        } else { return "No aporta cadenas válidas"; }
    } //Cierre del método
}
```

```
//Método que combina las cadenas aunque contengan menos de 3 caracteres (nuevo método)
String CombinadoSiempre (String valor_texto1, String valor_texto2) {
    int longitud_texto1 = 3; //Si podemos extraeremos tres caracteres
    int longitud_texto2 = 3;
    String CombinadoSiempre = ""; //Cadena vacía, variable local
    if (valor_texto1.length() < 3) { //Si hay menos de 3 caracteres extraemos los que haya
        longitud_texto1 = valor_texto1.length();
    }
    if (valor_texto2.length() < 3) { longitud_texto2 = valor_texto2.length(); }
    CombinadoSiempre = valor_texto1.substring (0,longitud_texto1)+" "+ valor_texto2.substring (0, longitud_texto2);
    return CombinadoSiempre;
} //Cierre del método

} //Cierre de la clase
```

Recuerda que **los espacios cuentan**. No es lo mismo una cadena de longitud cero o cadena vacía, representada por dos comillas sin espacio entre ellas, que una cadena que contenga un espacio, representada por dos cadenas que contengan un espacio entre ellas, cuya longitud o número de caracteres es 1.

EJERCICIO

Crea un objeto de tipo Combinador y prueba a ejecutar los dos métodos disponibles pasando distintos parámetros: cadenas con más de tres caracteres, una cadena con más de tres y otra con menos de tres y ambas con menos de tres caracteres. Fíjate en el resultado que ofrece el método CombinadoSiempre cuando se le pasan como parámetros dos cadenas vacías. ¿Te parece lógico el resultado?

En la clase tenemos definidos dos atributos o campos. ¿Tienen utilidad conocida a la vista del código de la clase? ¿Se usan en los métodos que tenemos definidos?

Puedes comprobar si es correcta tu respuesta consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00651B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

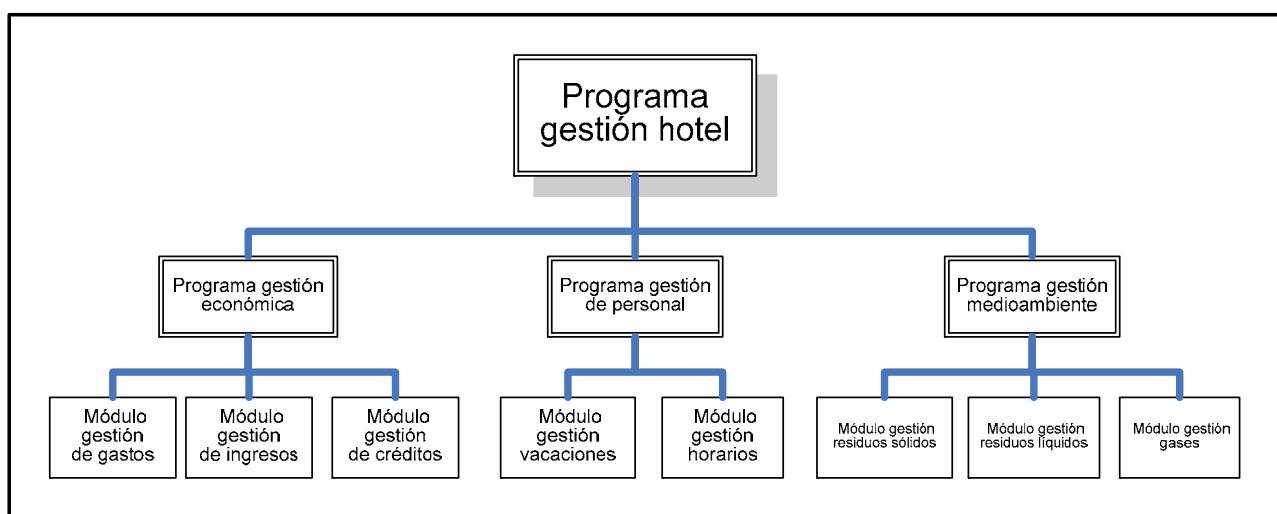
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONCEPCIÓN DE PROGRAMAS EN JAVA MEDIANTE ABSTRACCIÓN Y MODULARIZACIÓN

Los programas informáticos suelen ser muy complejos como para abordarlos en su conjunto. Por ello han de usarse técnicas para dividir el problema (estrategia “divide y vencerás”) que nos permitan crear programas complejos a partir de partes más simples. Hablamos de abstracción para referirnos a partes complejas ensambladas a partir de partes simples.



Supongamos que necesitamos un programa informático para la gestión de un hotel. El programa habrá que dividirlo para construirlo y ensamblarlo para su puesta en funcionamiento:



El programador que se encarga de una parte del software usa otras partes de software desarrolladas por otros sin analizar sus detalles (se abstraen de los detalles).

Hablamos de modularización del software en alusión a **dividir un programa en partes independientes que pueden ser construidas y probadas por separado** para luego ensamblarlas formando un todo. De acuerdo con esta terminología “abstraer” sería subir de nivel o ensamblar, mientras que modularizar sería bajar de nivel o despiezar.

En programación orientada a objetos la abstracción la materializamos construyendo objetos que usan o son combinación de otros objetos más simples. De este modo podemos construir objetos de gran complejidad sin necesidad de crear un código largo y complejo.

Próxima entrega: CU00652B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

UN EJEMPLO DE CÓDIGO JAVA BÁSICO. CREAR CLASES CON CAMPOS, CONSTRUCTOR Y MÉTODOS

Para familiarizarnos con el código Java escribe, ejecuta y estudia el código que mostramos a continuación, correspondiente a dos clases. Todos los elementos que forman parte de él ya los hemos estudiado excepto la llamada *this* (*0, 0, ""*). La palabra clave *this* tiene distintos usos en Java y en general podríamos interpretarla como “este objeto”.



La invocación *this*, o *this (parámetros)* supone una invocación al constructor que coincide con los parámetros que se pasan para que se execute. Al igual que existen formas de invocar a métodos, existen formas de invocar a constructores, y ésta es una de ellas.

El código de la primera clase sería el siguiente:

```
/* Esta clase representa un depósito cilíndrico donde se almacena aceite */

public class Deposito {

    //Campos de la clase

    private float diametro;

    private float altura;

    private String idDeposito;

    //Constructor sin parámetros auxiliar
    public Deposito () { //Lo que hace es llamar al constructor con parámetros pasándole valores vacíos
        this(0,0,"");
    } //Cierre del constructor

    //Constructor de la clase que pide los parámetros necesarios
    public Deposito (float valor_diametro, float valor_altura, String valor_idDeposito) {
        if (valor_diametro > 0 && valor_altura > 0) {
            diametro = valor_diametro;
            altura = valor_altura;
            idDeposito = valor_idDeposito;
        } else {
            diametro = 10;
            altura = 5;
            idDeposito = "000";
            System.out.println ("Creado depósito con valores por defecto diametro 10 metros altura 5 metros id 000" );
        }
    } //Cierre del constructor
}
```

```

public void setValoresDeposito (String valor_idDeposito, float valor_diametro, float valor_altura) {
    idDeposito = valor_idDeposito;
    diametro = valor_diametro;
    altura = valor_altura;
    if (idDeposito != "" && valor_diametro > 0 && valor_altura > 0) {
    } else {
        System.out.println ("Valores no admisibles. No se han establecido valores para el depósito");
        //Deposito (0.0f, 0.0f, ""); Esto no es posible. Un constructor no es un método y por tanto no podemos llamarlo
        idDeposito = "";
        diametro = 0;
        altura = 0;
    } } //Cierre del método

public float getDiametro () { return diametro; } //Método de acceso
public float getAltura () { return altura; } //Método de acceso
public String getIdDeposito () { return idDeposito; } //Método de acceso
public float valorCapacidad () { //Método tipo función
    float capacidad;
    float pi = 3.1416f; //Si no incluimos la f el compilador considera que 3.1416 es double
    capacidad = pi * (diametro/2) * (diametro/2) * altura;
    return capacidad;
} } //Cierre de la clase

```

En el método `setValoresDeposito` nos encontramos un código un tanto extraño: un `if` donde las instrucciones a ejecutar se encuentran vacías. Esto es admitido en Java, tanto en un `if` como en un `else` o en otras instrucciones. En este caso, el código equivale a: "Si el `idDeposito` es distinto de una cadena vacía y el `valor_diametro` es mayor que cero y el `valor_altura` es mayor que cero no se hace nada, y en caso contrario se han de ejecutar las instrucciones indicadas en el `else`". Este tipo de construcciones no consideramos conveniente utilizarlas frecuentemente. Tan solo pueden ser indicadas cuando queremos remarcar que en determinadas circunstancias no se debe ejecutar ninguna instrucción.

Otra cuestión a tener en cuenta es que de momento estamos desarrollando una programación informal: el sistema de comentarios no se atiene a lo establecido por el sistema de documentación de Java, y hemos incluido algunas sentencias de impresión por consola que normalmente no forman parte del código de los programas. Usaremos estas y otras técnicas informales con el fin de facilitar el aprendizaje, no porque puedan ser recomendadas como técnicas de programación.

La segunda clase sería la siguiente:

```

/*Esta clase representa un conjunto de depósitos formado por entre 2 y 3 depósitos */
public class GrupoDepositos {
    //Campos de la clase, algunos de ellos son tipo objetos de otra clase
    private Deposito deposito1;
    private Deposito deposito2;
    private Deposito deposito3;
    private String idGrupo;
    private int numeroDepositosGrupo;
}

```

```
//Constructor para la clase. En ella se crean objetos de otra clase.
public GrupoDepositos (int numeroDeDepositosGrupo, String valor_idGrupo) {
    idGrupo = valor_idGrupo;
    switch (numeroDeDepositosGrupo) {
        case 1: System.out.println ("Un grupo ha de tener más de un depósito"); break;

        case 2:
            deposito1 = new Deposito(); /*Al crear el objeto automáticamente se llama al constructor del mismo, en este caso sin parámetros. ESTO ES EJEMPLO DE SINTAXIS DE CREACIÓN DE UN OBJETO, EN ESTE CASO DENTRO DE OTRO */
            deposito2 = new Deposito();
            numeroDepositosGrupo = 2;
            break;

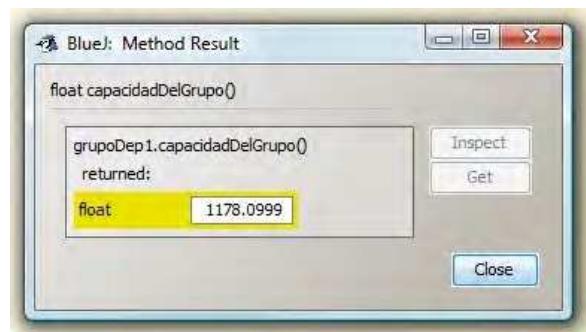
        case 3: deposito1 = new Deposito(); deposito2 = new Deposito(); deposito3 = new Deposito();
            numeroDepositosGrupo = 3;
            break;

        default: System.out.println ("No se admiten más de tres depósitos");
            //Esto no evita que se cree el objeto.
            break;
    } //Cierre del switch
} //Cierre del constructor

public int getNumeroDepositosGrupo () { return numeroDepositosGrupo; }

public String getIdGrupo () { return idGrupo; }
public float capacidadDelGrupo () { //Este método usa objetos de otra clase e invoca métodos de otra clase
    if (numeroDepositosGrupo == 2) { return (deposito1.valorCapacidad() + deposito2.valorCapacidad()); }
    else { return (deposito1.valorCapacidad() + deposito2.valorCapacidad() + deposito3.valorCapacidad()); }
    //Si el grupo se ha creado con un número de depósitos distinto de 2 o 3 saltará un error en tiempo de ejecución
} //Cierre del método
} //Cierre de la clase
```

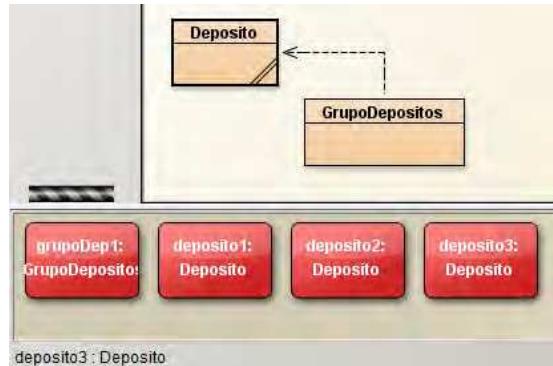
Con botón derecho sobre el ícono de la clase GrupoDepositos, crea un grupo de depósitos que conste de 3 depósitos y cuyo idGrupo sea “Grupo KHP”. Invoca los métodos que devuelven el número de depósitos del grupo, el identificador del grupo y la capacidad de los depósitos del grupo. Como capacidad deberás obtener un valor de aproximadamente 1178.1 unidades cúbicas.



Verifica con la calculadora si este valor es correcto y trata de razonar sobre por qué se obtiene este valor y no otro. Crea también distintos objetos de tipo Deposito y utiliza sus métodos. En el caso de

resultados numéricos, comprueba si los resultados que te ofrece el ordenador son correctos comparándolos con los resultados que te ofrece una calculadora.

Este ejemplo de código, todavía muy elemental y rudimentario, tiene un diagrama de clases donde nos indica que la clase GrupoDepositos usa a la clase Deposito.



A modo de resumen, el siguiente esquema nos indica lo que podemos hacer con este código. De este ejemplo lo único que nos interesa es practicar cómo una clase puede usar objetos y métodos de otra clase y la sintaxis a emplear. Aunque no hemos creado ningún programa, estamos viendo cómo crear clases y objetos que intervendrán en los programas.

¿Qué podemos hacer con este código?

Crear objetos Deposito	Especificando diámetro, altura e id Con unos valores de diámetro, altura e id por defecto	Y	Consultar altura Consultar diámetro Consultar idDeposito Establecer valores de altura, diámetro e id Calcular capacidad en volumen = $\pi * R^2 * H$
Crear objetos GrupoDeposito	Formados por 2 ó 3 depósitos con dimensiones e id por defecto (iguales para todos)	Y	Consultar la id del grupo Consultar el número de depósitos del grupo Calcular la capacidad del grupo

Próxima entrega: CU00653B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONCEPTO O DEFINICIÓN DE MÉTODO INTERNO Y MÉTODO EXTERNO EN JAVA

Cuando programamos una clase en Java es frecuente que usemos dos tipos de llamadas a métodos. Un tipo de llamada es aquella que hacemos a un método de la propia clase. Otro tipo es aquella que hacemos a un método de otra clase. En base a ello hacemos estas definiciones:



Llamada a método interno: es aquella llamada a un método ubicado en la misma clase en que se produce la llamada. La sintaxis es: nombreDelMétodo (parámetros si los hay);. Ejemplo: calcularCoste();.

Llamada a método externo: cuando una clase usa objetos de otra clase y se hacen llamadas a métodos de ese objeto (métodos definidos en otra clase) decimos que se llama a un método externo. La sintaxis es: nombreDelObjeto.nombreDelMétodo (parámetros si los hay);. Ejemplo: deposito1.calcularCoste();.

La llamada a métodos externos usa la notación de punto, es decir, nombre del objeto [punto] nombre del método. En la llamada a métodos internos esta notación se omite. Se sobreentiende que al llamar a un método el objeto al que se refiere es el que se define en la clase.

EJERCICIO

Reflexiona y responde a las siguientes preguntas:

- ¿Es posible llamar a un método externo de un objeto que ha sido declarado pero no creado?
- ¿Es posible llamar a un método externo de un objeto “A” que ha sido declarado y creado pero cuyo constructor está vacío?
- Una llamada al método pow de la clase Math que hacemos dentro de un método definido en una clase ¿Es una llamada a un método interno o a un método externo? ¿La llamada al método pow se hace sobre un objeto declarado y creado o no declarado y no creado?

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00654B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PALABRA CLAVE THIS EN JAVA. CONTENIDO NULL POR DEFECTO DE UN OBJETO.

Ya hemos visto en el epígrafe anterior que la palabra clave *this* puede ser usada para invocar a un constructor. Sin embargo, su uso quizás más frecuente en Java tiene lugar en otro contexto: cuando existe sobrecarga de nombres. La sobrecarga de nombres se da cuando tenemos una variable local de un método o constructor, o un parámetro formal de un método o constructor, con un nombre idéntico al de un campo de la clase.



Este sería un mal ejemplo de sobrecarga de nombres, tanto con parámetros como con variables locales:

```
public class Mensaje {
    //Campos
    private String remitente;
    private String para;
    private String texto;

    //Constructor con sobrecarga de nombres al coincidir nombres de parámetros con los de campos
    public Mensaje (String remitente, String para, String texto) {
        remitente = remitente; //¿Cómo va a saber Java cuál es el parámetro y cuál el campo?
        para = para; //¿Cómo va a saber Java cuál es el parámetro y cuál el campo?
        texto = texto; //¿Cómo va a saber Java cuál es el parámetro y cuál el campo?
    } //Cierre del constructor

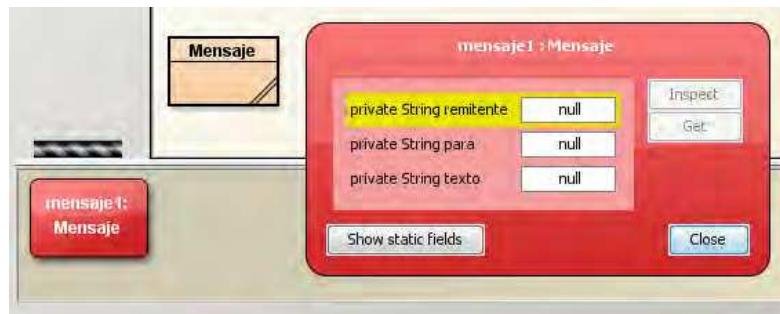
    //Método con sobrecarga de nombres al coincidir un parámetro con un campo
    public void setRemitente (String remitente) {
        remitente = remitente; //¿Cómo va a saber Java cuál es el parámetro y cuál el campo?
    } //Cierre del método

    //Método con sobrecarga de nombres al coincidir una variable local con un campo
    public void extraerFraccionTexto () {
        String texto = ""; //Esto supone declarar una variable local que "tapa" al campo
        texto = texto.substring (0, 5); //¿Cómo va a saber Java si nos referimos al campo?
    } //Cierre del método
} // Cierre de la clase
```

Escribe y compila el código anterior. El código, efectivamente compila. ¿Por qué? Porque Java tiene previstos mecanismos para resolver conflictos de nombres y aplica una reglas. En concreto, la regla de que “un nombre hace referencia a la variable más local de entre las disponibles”. Y el carácter de local se interpreta de la siguiente manera:

Variante local > Parámetro formal > Campo de clase

Crea ahora un objeto de tipo Mensaje e introduce un texto como *remitente*, por ejemplo “Juan”, otro como *para*, por ejemplo “Pedro” y otro como *texto*, por ejemplo “Saludos desde Buenos Aires”. A continuación, inspecciona el estado del objeto. El resultado será algo así:



Interpretaremos ahora por qué nos aparece *null* como contenido de los atributos del objeto. En el constructor hemos definido tres parámetros: *remitente*, *para* y *texto*. Luego hemos indicado que *remitente = remitente*; ¿Qué variable usa Java? Tiene que elegir entre usar el campo o usar el parámetro del método. No puede usar ambos porque no podría saber cuándo usar uno y cuándo usar otro. El conflicto lo resuelve utilizando el parámetro del método, es decir, interpretando que “el parámetro es igual al parámetro”. Esto no tiene ningún efecto, lo que significa que el atributo *remitente* se queda sin inicializar. Lo hemos declarado, pero no lo hemos inicializado.

En nuestra clase tenemos tres campos o variables de instancia cuyo ámbito es toda la clase. El ámbito de los parámetros o variables locales es exclusivamente el constructor o método al cual se aplican.

Recordemos que un String es un objeto en Java. Un objeto no inicializado carece de contenido y esto nos lo informa Java indicándonos un contenido aparente *null*. La palabra clave *null* indica que un objeto se encuentra vacío, carente de contenido.

La palabra clave null indica que un objeto se encuentra vacío, carente de contenido. Esto puede deberse a que no se ha inicializado, a que su contenido ha sido eliminado usando una instrucción del tipo nombreDelObjeto = null;, o a otros motivos. En el caso de un String, hay que diferenciar entre que el objeto tenga como contenido una cadena vacía, que al fin y al cabo es un contenido, o que carezca de contenido. La palabra clave null no es aplicable a los tipos primitivos, que no admiten una asignación del tipo nombreVariable = null. Un tipo primitivo no inicializado tendrá como contenido un valor en el rango de valores admisibles para el tipo como 0 para int o false para boolean. Recordar que por norma inicializaremos siempre cualquier objeto o variable de forma explícita.

Una instrucción del tipo `if (remitente == null) { ... }` nos puede servir para detectar si un objeto ha sido inicializado adecuadamente.

Volvamos ahora al código de nuestra clase `Mensaje`. El conflicto de nombres vamos a solventarlo haciendo uso de la palabra clave `this`. Escribiendo `this.nombreDelCampo` le indicaremos a Java que nos referimos al atributo de la clase en vez de a una variable local o parámetro. Veámoslo aplicado en el código:

```
public class Mensaje {
    private String remitente;
    private String para;
    private String texto;

    //Constructor con sobrecarga de nombres al coincidir nombres de parámetros con los de campos
    public Mensaje (String remitente, String para, String texto) {
        this.remitente = remitente; //this.remitente es el campo y remitente el parámetro
        this.para = para; //this.para es el campo y para el parámetro
        this.texto = texto; //this.texto es el campo y texto el parámetro
    }

    //Método con sobrecarga de nombres al coincidir un parámetro con un campo
    public void setRemitente (String remitente) {
        this.remitente = remitente; //this.remitente es el campo y remitente el parámetro
    }

    //Método con sobrecarga de nombres al coincidir una variable local con un campo
    public String extraerFraccionTexto () {
        String texto = ""; //texto es una variable local
        texto = this.texto.substring (0, 5); //this.texto es el campo de los objetos de la clase
        return texto;
    }
}
```

Crea ahora un objeto de tipo `Mensaje` e inicialízalo introduciendo valores para los parámetros requeridos por el constructor. El resultado es que ahora sí se produce una inicialización correcta porque hemos definido adecuadamente cómo han de gestionarse los nombres de variables.



En resumen, **usando la palabra clave `this` podemos evitar que los parámetros o variables locales tapen a las variables globales**. Para interpretar el significado de `this`, podemos pensar que hace referencia al “objeto actual” en un momento dado. Repetir los nombres de campos como parámetros y como variables locales queda a elección de cada programador. En sí misma, esta práctica no puede considerarse ni buena ni mala siempre que se mantenga una coherencia y lógica global de nombres. La

realidad es que la repetición de nombres es muy frecuente en la práctica de la programación porque a la hora de desarrollar programas largos intervienen muchas variables y la búsqueda de nombres distintos puede hacer perder tiempo al programador y hacer más difícil de seguir el código.

EJERCICIO

Define una clase Profesor considerando los siguientes atributos de clase: nombre (String), apellidos (String), edad (int), casado (boolean), especialista (boolean). Define un constructor que reciba los parámetros necesarios para la inicialización y otro constructor que no reciba parámetros. El nombre de los parámetros debe ser el mismo que el de los atributos y usar this para asignar los parámetros recibidos a los campos del objeto. Crea los métodos para poder establecer y obtener los valores de los atributos, con sobrecarga de nombres y uso de this en los métodos setters (los que sirven para establecer el valor de los atributos). Compila el código para comprobar que no presenta errores, crea un objeto usando el constructor con sobrecarga. Comprueba que se inicializa correctamente consultando el valor de sus atributos después de haber creado el objeto. Usa los métodos setters y comprueba que funcionen correctamente.

Para comprobar si es correcta tu solución puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00655B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

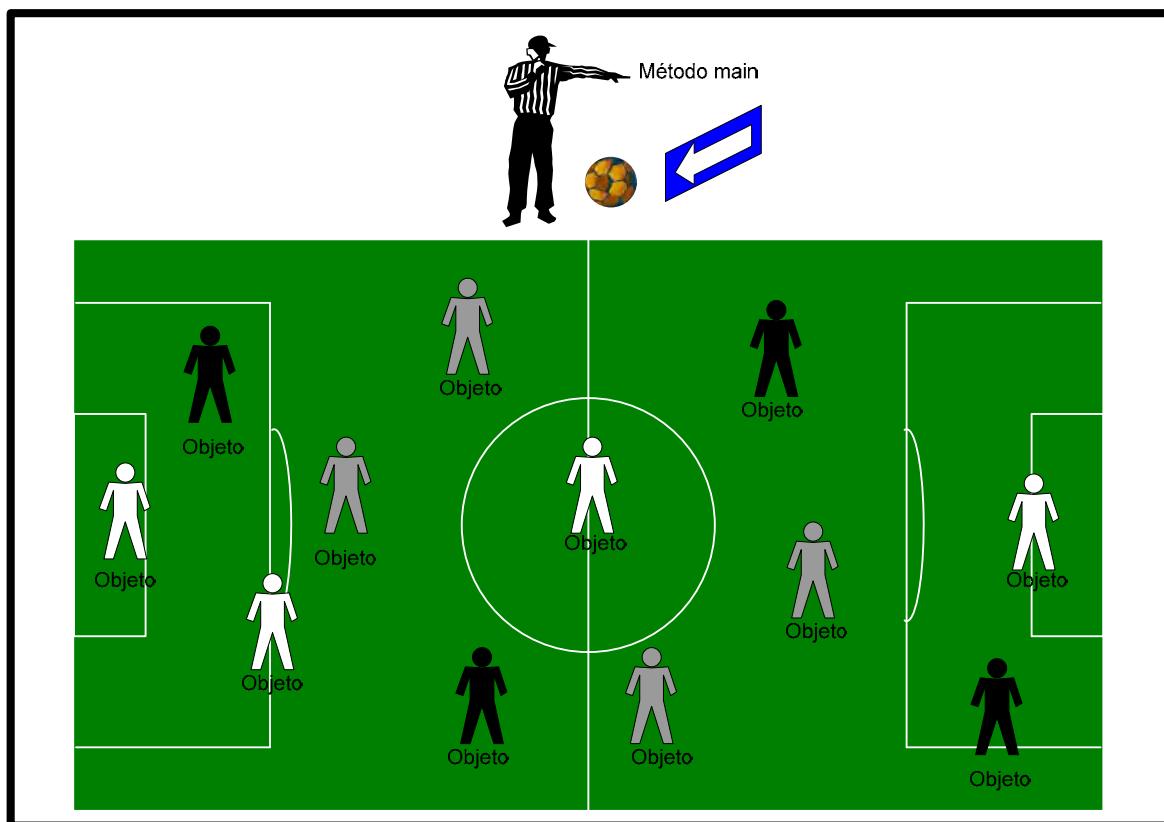
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CLASE CON EL MÉTODO MAIN: CLASE PRINCIPAL, INICIADORA O “PROGRAMA PRINCIPAL”

Hasta ahora hemos visto código implementando clases y que las clases definen tipos, es decir, nos permiten crear objetos del tipo definido por la clase. Pero a todas estas, ¿dónde está el programa? Todavía no hemos visto ningún programa, y ya es hora de que abordemos este asunto.



Sabemos que los métodos en Java pueden tener cualquier nombre (excluido el de palabras clave). Existe un nombre de método que está reservado en Java y otros lenguajes: el método *main*. Este método es un método especial en tanto en cuanto es el que da lugar al inicio del programa. Si comparamos un programa con un partido de fútbol, el método main sería el responsable de poner el partido en juego.

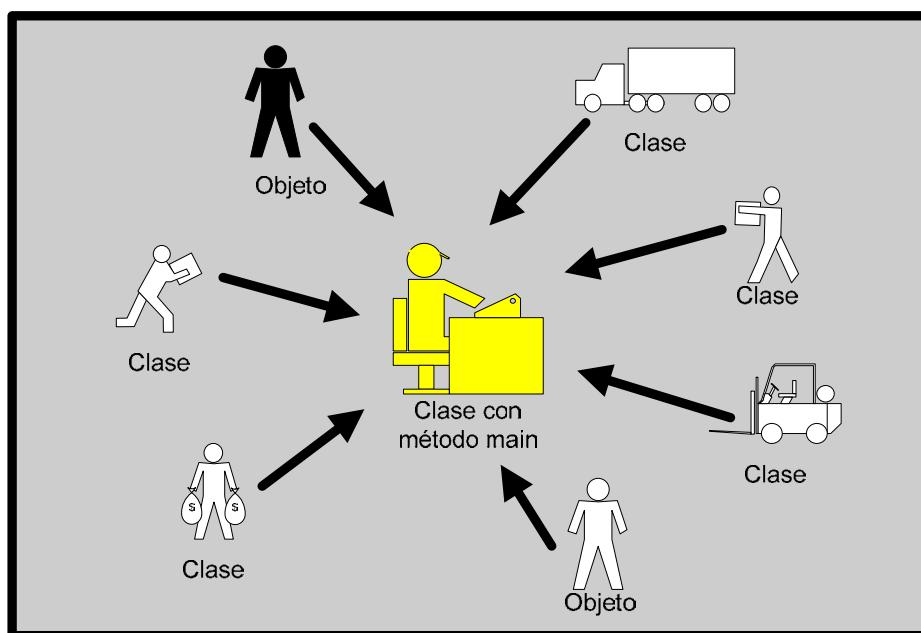


Es importante tener claro que **el método main no es el elemento principal en el desarrollo del programa**. El programa, de acuerdo con el paradigma de programación orientada a objetos, se desarrolla mediante la interacción entre objetos, que en la figura hemos representado como jugadores en el campo de fútbol. Por tanto el cometido del método main normalmente es iniciar el programa

(poner el balón en juego) y permanecer en un segundo plano mientras los objetos interactúan entre sí, controlando el desarrollo de la situación como si del árbitro se tratara. El método main normalmente no será muy extenso en cuanto a líneas de código respecto al resto del código (clases). Si esto ocurriera, posiblemente sería indicador de que este método tiene más protagonismo del que debe, y en nuestro símil el árbitro no debe ser protagonista del partido. En algunos ejemplos de código que desarrollemos quizás la clase con el método main contenga gran parte del código total, pero si esto es así será porque lo estamos utilizando con fines didácticos. En programas profesionales esto no debe ocurrir.

El método main es indudablemente importante. Por eso y por motivos históricos en el desarrollo de la programación muchas veces se alude a él (o a la clase donde se sitúa) como clase principal o “programa” principal. Pero hay que tener bien claro que su carácter principal se debe a que **es quien inicia el desarrollo del programa**, no a que sea quien debe asumir el protagonismo del mismo.

Un error que cometen frecuentemente las personas que están aprendiendo Java es suponer que el método main es la parte principal del programa, en el sentido de que debe concentrar los procesos importantes y la mayor parte del código. Esta concepción errónea se reflejaría en el siguiente esquema, donde las clases y objetos están al servicio del método main (o de la clase donde se encuentre) que es quien centraliza las operaciones. **Este esquema es posible, pero no es el adecuado** dentro del estilo de programación orientada a objetos y no aprovecha todo el potencial de esta forma de programación.



Próxima entrega: CU00656B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

SINTAXIS Y CÓDIGO EJEMPLO DE USO DEL MÉTODO MAIN

El método main de momento lo situaremos en una clase independiente destinada exclusivamente a contener este método, aunque esto no es obligatorio: la clase con el método main podría tratarse como una clase más y el método main como un método más. Nosotros preferiremos diferenciarlo por motivos didácticos.



La sintaxis que emplearemos para el método main será la siguiente:

```
public static void main (String [ ] args) {
    //Aquí las instrucciones del método
}
```

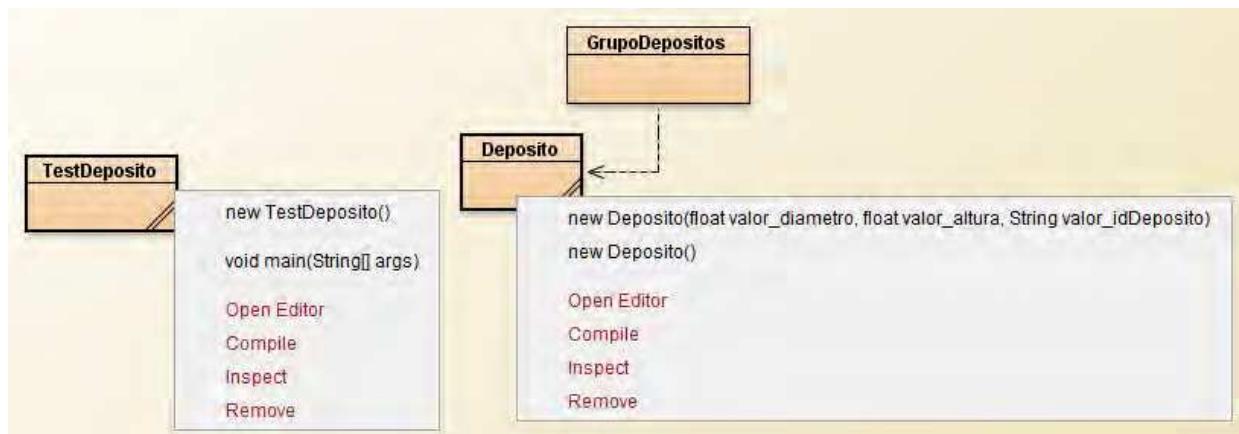
No vamos a entrar ahora a detallar el significado de los términos de la sintaxis: la iremos entendiendo a medida que avancemos. Digamos que la clase con el método main es especial porque podemos invocar al método main sin necesidad de crear antes un objeto de la clase. En las clases “normales” no podemos invocar ningún método si no hemos creado un objeto previamente.

Cuando creamos un programa para chequear el funcionamiento de otras clases a modo de prueba es frecuente ponerle como nombre TestNombreDelPrograma. Nosotros vamos a suponer que vamos a trabajar con el código de las clases Deposito y GrupoDepositos que habíamos visto en un ejemplo anterior, por lo que vamos a llamar a la clase del programa principal TestDeposito. Para empezar, creamos una clase y escribimos esto:

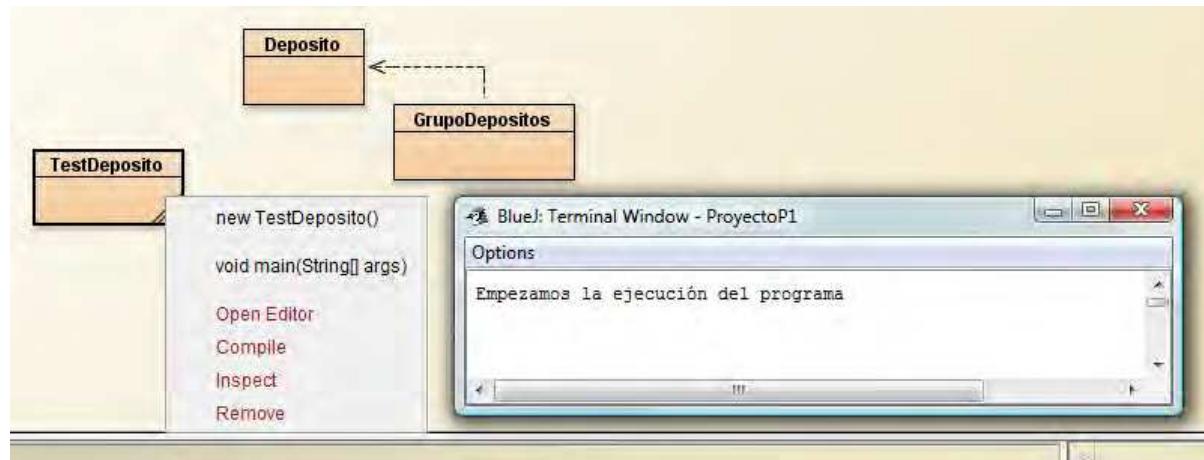
```
// Clase principal iniciadora del programa ejemplo aprenderaprogramar.com
public class TestDeposito {
    public static void main (String [ ] args) {
        //Aquí las instrucciones de inicio y control del programa
        System.out.println ("Empezamos la ejecución del programa");

    } //Cierre del main
} //Cierre de la clase
```

¿Qué diferencia a la clase iniciadora de otras clases? En primer lugar que contiene el método main y este es ejecutable sin necesidad de crear un objeto. En BlueJ lo visualizamos así:



Mientras que con Deposito, que es una clase “normal”, únicamente podemos invocar a los constructores de la clase para crear objetos, en TestDeposito podemos invocar al método main que dará lugar a la ejecución del código contenido en el mismo. Si lo hacemos, el resultado obtenido será el siguiente:



Ejecuta el código de la clase con método main que hemos escrito anteriormente y comprueba que obtienes el resultado esperado (que se muestre por pantalla un mensaje).

Próxima entrega: CU00657B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PEDIR DATOS POR CONSOLA (TECLADO) EN JAVA.

Vamos a crear un programa elemental para pedir datos por consola (entrada de teclado del usuario) y para ello vamos a basarnos en una clase del API de Java: la clase Scanner. Estudia el siguiente código y busca la documentación sobre la clase. Trata de verificar que los métodos que empleamos aparecen en la documentación y consulta su detalle.



Debes acostumbrarte a buscar en el API, leer documentación y utilizar clases y métodos disponibles. Cuando programes en Java tendrás que hacerlo con frecuencia, como vamos a hacer ahora con la clase Scanner. Escribe este código en tu entorno de desarrollo.

```
import java.util.Scanner; //Importación del código de la clase Scanner desde la biblioteca Java
//Código de nuestra clase principal ejemplo aprenderaprogramar.com
public class TestPrograma1 {

    public static void main (String [ ] args) {
        System.out.println ("Empezamos el programa");
        System.out.println ("Por favor introduzca una cadena por teclado:");
        String entradaTeclado = "";
        Scanner entradaEscaner = new Scanner (System.in); //Creación de un objeto Scanner
        entradaTeclado = entradaEscaner.nextLine (); //Invocamos un método sobre un objeto Scanner
        System.out.println ("Entrada recibida por teclado es: \"" + entradaTeclado + "\"");
    } //Cierre del main
} //Cierre de la clase
```

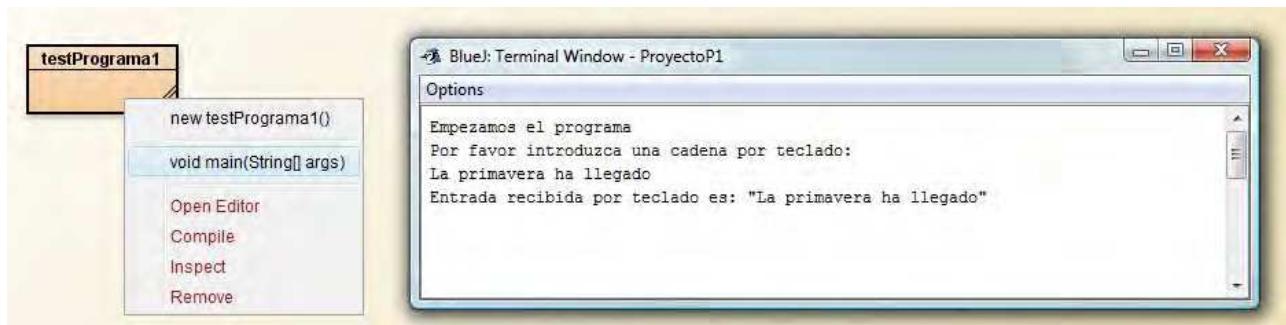
El constructor que hemos utilizado dentro de los varios disponibles en la clase Scanner es el que tiene la firma Scanner (`InputStream source`). Le pasamos como parámetro `System.in`. Ahora no debemos preocuparnos por comprender los detalles, simplemente hemos de entender que estamos creando un objeto de un tipo definido por el API de Java, usando un constructor al que le pasamos un parámetro.

Cuando sobre el objeto Scanner se invoca el método `nextLine()`, se obtiene un `String` que se corresponde con el texto que haya introducido el usuario mediante el teclado. Este resultado del método lo almacenamos en la variable `entradaTeclado`. Finalmente, mostramos por pantalla cuál ha sido el contenido del texto recibido por teclado. En la sentencia en que imprimimos el mensaje por pantalla hemos usado el carácter `\`. Este carácter, una **barra invertida o backslash**, no se muestra en pantalla y es interpretado por Java como indicación de que el siguiente carácter forma parte del texto. Esta es la forma que tenemos para incluir unas comillas en un texto que queramos mostrar por pantalla. Sin este escape, cualquier comilla se interpretaría como delimitadora de una cadena. Otra aplicación del backslash es la de forzar un salto de línea. Hasta ahora hemos trabajado invocando el método `println` del objeto `System.out` para imprimir una línea en la ventana de consola. Cabe citar que

otro método disponible para System.out es print, con el resultado de que tras un print no se inserta un cambio de línea y retorno de carro como ocurre con los println, por lo que los textos aparecen “uno a continuación de otro”. Esto es útil, por ejemplo para escribir listas de números separados por comas o guiones sin necesidad de usar una línea por número.

Ahora bien, si usamos print normalmente en el último elemento que queremos mostrar sí querremos insertar un salto de línea y retorno de carro. Esto lo podemos hacer con el carácter de escape '\n'. Un ejemplo sería: System.out.print ("Esta es una primera frase. \nY esta es una segunda frase.\n");

Volvamos al código que habíamos escrito. Ejecuta el método main de la clase TestPrograma1.



Nuestra clase TestPrograma1 define un tipo, pero en realidad no nos interesa instanciar esta clase. La utilizaremos como un paquete de código que podemos ejecutar.

Hay que incidir en una cuestión que ya hemos comentado. ¿Qué sentido tiene que una entrada de teclado sea un objeto? Si no lo tienes claro en vez de seguir avanzando vuelve a leer los epígrafes anteriores relacionados con objetos.

Hay distintas maneras de pedir entradas por teclado en Java. Seguiremos viendo cuestiones relacionadas con este asunto en próximos apartados.

EJERCICIO

Crea una clase con un método main que pida una entrada de teclado y usando condicionales, el método length de la clase String y el método substring de la clase String, muestre un mensaje indicando:

- Si la cadena introducida tiene menos de 5 caracteres, entre 5 y 15 caracteres o más de 15 caracteres.
- Si la cadena introducida comienza por la letra a.

Ejemplo: se pide la entrada y el usuario introduce “vereda”. Por pantalla se debe mostrar: “La cadena introducida tiene entre 5 y 15 caracteres y no comienza por a”.

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00658B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

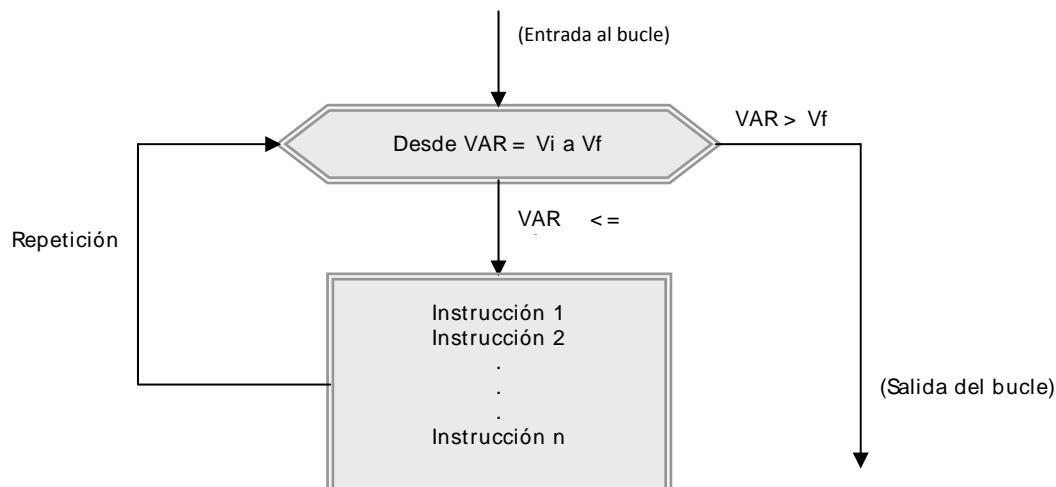
CONCEPTO GENERAL DE BUCLE

Nos referimos a estructuras de repetición o bucles en alusión a instrucciones que permiten la repetición de procesos un número n de veces. Los bucles se pueden materializar con distintas instrucciones como for, while, etc. Un bucle se puede anidar dentro de otro dando lugar a que por cada repetición del proceso exterior se ejecute n veces el proceso interior. Lo veremos con ejemplos.



BUCLE CON INSTRUCCIÓN FOR. OPERADOR ++ Y --. SENTENCIA BREAK.

En Java existen distintas modalidades de for. El caso más habitual, que es el que expondremos a continuación, lo denominaremos for normal o simplemente for. Conceptualmente el esquema más habitual es el siguiente:

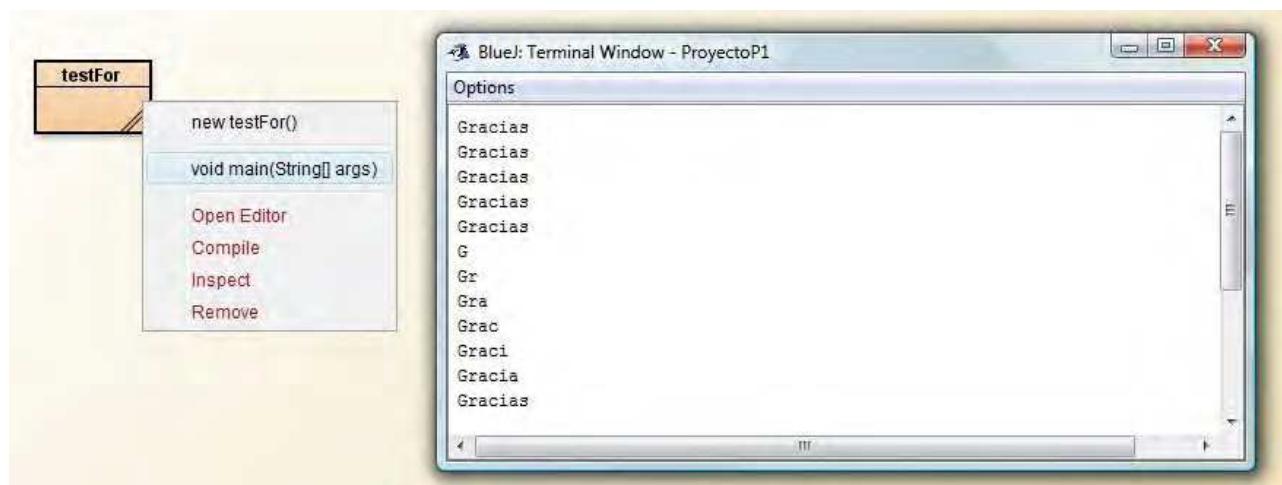


La sintaxis habitual es: `for (int i = unNúmero; i < otroNúmero; i++) { instrucciones a ejecutarse }`, donde `int i` supone la declaración de una variable específica y temporal para el bucle. El nombre de la variable puede ser cualquiera, pero suelen usarse letras como `i`, `j`, `k`, etc. `unNúmero` refleja el número en el que se empieza a contar, con bastante frecuencia es `0` ó `1`. `i < otroNúmero` ó `i <= otroNúmero` refleja la condición que cuando se verifique supondrá la salida del bucle y el fin de las repeticiones. `i++` utiliza el operador `++` cuyo significado es “incrementar la variable `i` en una unidad”. Este operador se puede usar en cualquier parte del código, no es exclusivo para los bucles for. Igualmente se dispone del operador “gemelo” `--`, que realiza la operación en sentido contrario: reduce el valor de la variable en una unidad. Escribe el siguiente código en tu entorno de desarrollo, ejecuta el método main de esta clase y comprueba los resultados:

```
//Clase test del for ejemplo aprenderaprogramar.com
public class testFor {

    public static void main (String [ ] args) {
        for (int i = 0; i < 5; i++) { //Repite Gracias cinco veces
            System.out.println ("Gracias");
        }
        for (int i=0; i < ("Gracias").length(); i++) { //Va devolviendo en cada iteración una letra más de la cadena
            System.out.println ("Gracias".substring (0, i+1));
        }
    } //Cierre del main

} //Cierre de la clase
```



Un bucle for (o de cualquier otro tipo) puede ser interrumpido y finalizado en un momento intermedio de su ejecución mediante una instrucción break;. El uso de esta instrucción dentro de bucles solo tiene sentido cuando va controlada por un condicional que determina que si se cumple una condición, se interrumpe la ejecución del bucle.

EJERCICIO

Crea una clase con un método main que pida una entrada de teclado y usando un bucle for, el método length de la clase String y el método substring de la clase String, muestre cada una de las letras que componen la entrada. Por ejemplo si se introduce "ave" debe mostrar:

Letra 1: a

Letra 2: v

Letra 3: e

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

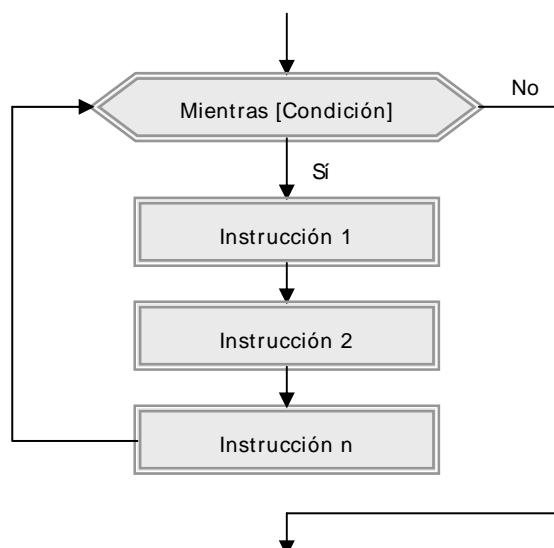
Próxima entrega: CU00659B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

BUCLE CON INSTRUCCIÓN WHILE EN JAVA. EJEMPLO USO DE BREAK.

El bucle while presenta ciertas similitudes y ciertas diferencias con el bucle for. La repetición en este caso se produce no un número predeterminado de veces, sino mientras se cumpla una condición. Conceptualmente el esquema más habitual es el siguiente:



La sintaxis en general es: `while (condición) { instrucciones a ejecutarse }` donde *condición* es una expresión que da un resultado true o false en base al cual el bucle se ejecuta o no. Escribe y prueba el siguiente código, donde además vemos un ejemplo de uso de la instrucción `break;`.

```

//Clase test del while curso aprenderaprogramar.com
public class testWhile {

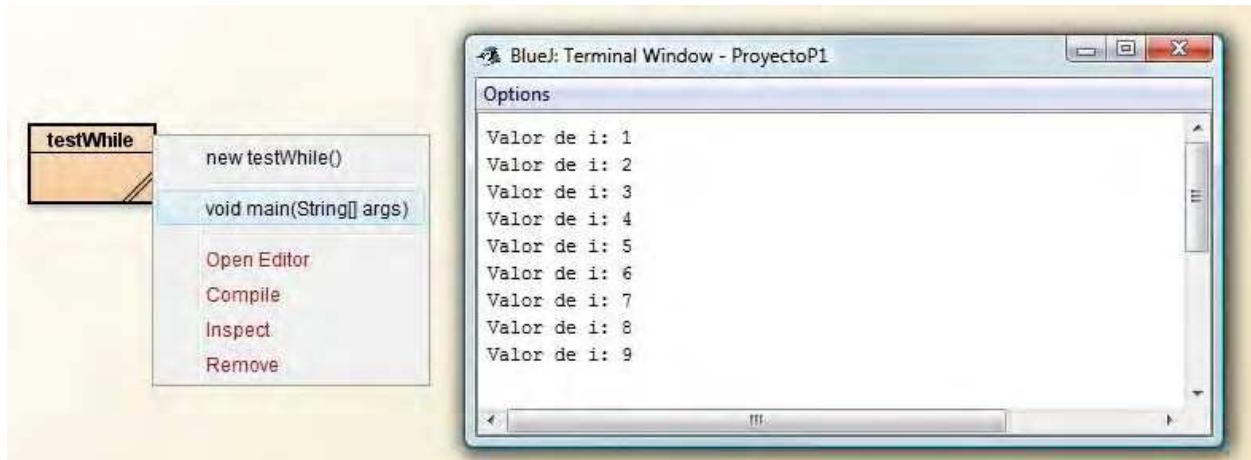
    public static void main (String [ ] args) {

        int i = 0;
        while (true) {          //Condición trivial: siempre cierta
            i++;
            System.out.println ("Valor de i: " + i);
            if (i==9) { break;}
        }

    } //Cierre del main

} //Cierre de la clase
  
```

En este código hemos hecho algo un poco extraño. Como condición a evaluar hemos puesto “true”. Esto significa que la condición es siempre verdadera, lo que en teoría daría lugar a un bucle infinito y a un bloqueo del ordenador. Sin embargo, utilizamos un contador auxiliar que inicializamos en cero y en cada repetición del bucle aumentamos en una unidad. A su vez, introducimos una condición dentro del bucle según la cual cuando el contador alcanza el valor 9 se ejecuta la instrucción break.



Este ejemplo debe valernos solo como tal: en general la condición de entrada al bucle será una expresión a evaluar como ($i < 10$ ó $a \geq 20$ ó reductor < compresor) y no un valor *true*. Y en general la salida a un bucle se realizará de forma natural mediante la evaluación de la condición y no mediante una instrucción *break*;

BUCLE CON INSTRUCCIÓN DO ... WHILE. EJEMPLO DE USO.

El bucle do ... while es muy similar al bucle while. La diferencia radica en cuándo se evalúa la condición de salida del ciclo. En el bucle while esta evaluación se realiza antes de entrar al ciclo, lo que significa que el bucle puede no llegar ejecutarse. En cambio, en un bucle do ... while, la evaluación se hace después de la primera ejecución del ciclo, lo que significa que el bucle obligatoriamente se ejecuta al menos en una ocasión. A modo de ejercicio, escribe este código y comprueba los resultados que se obtienen con él:

```
public class TestDelDoWhile { //Prueba del do ... while curso aprenderaprogramar.com
    public static void main (String [ ] Args) {
        int contador = 0 ;
        do { System.out.println ("Contando... " + (contador+1));
            contador += 1;
        } while (contador<10);  } }
```

EJERCICIO

Crea una clase con un método main que pida una entrada de teclado y usando **un bucle while**, el método length de la clase String y el método substring de la clase String, muestre cada una de las letras que componen la entrada. Por ejemplo si se introduce “ave” debe mostrar:

Letra 1: a

Letra 2: v

Letra 3: e

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00660B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EL DEBUGGER DE BLUEJ. DETENER UN PROGRAMA EN EJECUCIÓN

Muchas veces nos encontraremos con situaciones en las que un programa en ejecución se queda bloqueado. El caso típico puede ser el de entrada en un bucle infinito: un bucle donde nunca se cumple la condición de salida. Seguramente estés habituado a que un programa o sistema operativo se te bloquee ocasionalmente, obligándote a utilizar el administrador de tareas para finalizarlo, o incluso a reiniciar el ordenador.



Sin embargo, cuando ejecutamos un programa Java desde un entorno de desarrollo como BlueJ, normalmente existen opciones para detener la ejecución que tengamos corriendo en un momento dado sin tener que reiniciar y ni siquiera salir del entorno.

Llamamos “bug” a un error en un programa y *debugger* o *depurador* a un programa o extensión de un entorno de programación destinado al análisis de errores en los programas durante su desarrollo (pruebas) y a su corrección (debugging). Entre las funciones habituales de un debugger encontramos la ejecución paso a paso del programa y la posibilidad de establecer puntos de interrupción, es decir, puntos donde detenemos momentáneamente la ejecución del programa para analizar el estado de objetos o variables. BlueJ incorpora su propio depurador que podemos usar durante el desarrollo de nuestros programas. Además, el depurador nos servirá para detener un programa cuando ya no queramos continuar ejecutándolo, bien porque se haya bloqueado, bien porque queramos salir por cualquier motivo. El debugger de BlueJ tiene bastantes posibilidades pero solo vamos a comentar dos cuestiones:

- Poner un punto de interrupción en un programa:** esto supondrá que el programa se detendrá cuando se llegue a la línea donde hayas establecido el punto de interrupción. Para ello abre la clase donde quieras poner la interrupción y sitúate en la línea deseada. Compila la clase y en el menú Tools, elige la opción Set / Clear Breakpoint (o más sencillo aún, pulsa simplemente en la barra a la izquierda de la línea fuera del área de texto). Aparecerá un icono STOP en el lateral izquierdo que indica que se ha establecido el *breakpoint*.

```

public EntradaDeTeclado () { //Constructor
    entradaTeclado="";
    pedirEntrada(); //Esto es una llamada a un método interno. Al crear una entrada automáticamente ejecutamos el método
    //Cierre del constructor
}

public void pedirEntrada () { //Método de la clase
    Scanner entradaScanner = new Scanner (System.in);
    entradaTeclado = entradaScanner.nextLine ();
    // ...podríamos hacer System.out.println ("Entrada recibida");
    //Cierre del método pedirEntrada
}

public String getEntrada () { return entradaTeclado; } //cierre del método

public String getPrimerasPalabras () {
    ...
}

for (int i=0; i < entradaTeclado.length() - 1; i++) {
    if (entradaTeclado.substring(i, i+1).equals (" ")) {
        return entradaTeclado.substring(0, i+1);
    }
}
return entradaTeclado; //Caso de que no se detecte ningún espacio
//Cierre del método getPrimerasPalabras

public int getLongitud () { //Método de la clase
    return entradaTeclado.length();
} //Cierre del método getLongitud
//Cierre de la clase

```

Si modificas la clase el punto de interrupción desaparece automáticamente: solo se puede fijar sobre una clase compilada. Una vez ejecutes el código, cuando se alcance el breakpoint el programa se detendrá y se abrirá el debugger y la ventana de código en la línea donde se ha producido la interrupción.

La ventana del depurador también se puede abrir en cualquier momento, incluso durante la ejecución de un programa, desde la ventana principal de BlueJ en la opción de menú View → Show Debugger, aunque si no hay un programa en ejecución aparecerá vacía.

En una parada por punto de interrupción, en la ventana del depurador nos aparece información de utilidad. Por ejemplo, en *Call Sequence* los métodos que se están ejecutando en ese momento (si hubiera constructores en ejecución aparecerían como NombreDeLaClase.<init>). En *Instance variables* podemos ver cuál es el estado de las variables de instancia (campos) del objeto, mientras que en *Local variables* podemos ver cuál es el estado de las variables locales de los métodos en ejecución. Para tipos primitivos de Java y objetos String se nos mostrará directamente el contenido de la variable, mientras que en el caso de otros objetos nos indicará como contenido <object reference>, es decir, una referencia a un objeto.

Pulsando el botón *Step Into* podemos ir ejecutando el programa línea a línea y comprobando los valores de variables, métodos en ejecución, etc.

- b) **Detener la ejecución de un programa:** en cualquier momento se puede detener un programa, incluso si éste se ha bloqueado. Para ello vete a la ventana principal de BlueJ y abre el depurador. Seguidamente pulsa el botón señalado con un aspa roja y el texto *Terminate*. En la zona del debugger donde pone *Threads* te aparecerá un mensaje del tipo *main (finished)* que indica que la ejecución ha terminado completamente.

La terminación de un programa será algo a lo que tengas que recurrir con frecuencia, pues durante el desarrollo de código bien por errores que causan bloqueos o bien durante pruebas, es algo frecuente. En ocasiones te puedes encontrar con mensajes del tipo "Your program is already running. You cannot start another execution while the current one is still active...". Estos son debidos a que tratamos de ejecutar un programa cuando todavía hay otro en ejecución (esto puede suceder incluso aunque hayamos cerrado todas las ventanas y pensemos que el programa terminó). En estos casos, abre el depurador y fuerza la terminación del programa pulsando el botón Terminate.

Próxima entrega: CU00661B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PENSAR EN OBJETOS EN JAVA. UNA ENTRADA DE TECLADO COMO OBJETO.

En un programa anterior pedíamos una entrada por teclado al usuario (ver apartados anteriores). El caso visto corresponde a una ejecución propia de la programación estructurada: almacenamos una entrada de teclado en una variable, y hacemos algo con esa variable (en este caso mostrarla por pantalla). Revisa ese programa para refrescar los conceptos.



Ahora vamos a tratar de pensar como programadores bajo el paradigma de la programación orientada a objetos. En la programación orientada a objetos toda entidad que sea susceptible de tener unas propiedades y unos métodos puede ser definida en una clase. Si pensamos en una entrada de texto por teclado podríamos pensar en:

- a) **Propiedades:** tendrá un contenido (cadena de caracteres). Podríamos añadir otras.
- b) **Métodos:** podemos tener como operaciones definidas sobre una entrada de caracteres el pedir la entrada (pedirEntrada), el obtener el contenido de la entrada (getEntrada), el obtener la primera palabra de la entrada (getPrimeraPalabra), y el obtener la longitud como número de caracteres que tiene la entrada (getLongitud).

Con este razonamiento, cuando queramos pedir una entrada de teclado, crearemos un objeto EntradaDeTeclado sobre el cual podemos ejecutar operaciones diversas. Escribe esta definición de clase:

```

import java.util.Scanner; //Importación del código de la clase Scanner desde la biblioteca Java
// Definimos una clase EntradaDeTeclado aprenderaprogramar.com
public class EntradaDeTeclado {
    private String entradaTeclado; //Variable de instancia (campo) de la clase

    public EntradaDeTeclado () { //Constructor
        entradaTeclado="";
        pedirEntrada(); //Esto es una llamada a un método interno. Al crear un objeto automáticamente ejecutamos el método
    } //Cierre del constructor

    public void pedirEntrada () { //Método de la clase
        Scanner entradaEscaner = new Scanner (System.in);
        entradaTeclado = entradaEscaner.nextLine (); } //Cierre del método pedirEntrada

    public String getEntrada () { return entradaTeclado; } //Cierre del método getEntrada
    public String getPrimeraPalabra () {
        /*IMPORTANTÍSIMO: EN JAVA LOS OBJETOS NO SE PUEDEN COMPARAR USANDO == POR ESO ESTO NO FUNCIONARÁ
        if (entradaTeclado.substring(0,1)=="j") {
            System.out.println ("Hemos detectado una j");
        } else { System.out.println ("Mira esto:" + entradaTeclado.substring(0,1)); }*/
    }
}

```

Crear una entrada de teclado o consola en Java como objeto.

```

for (int i=0; i < entradaTeclado.length() - 1; i++) {
    if (entradaTeclado.substring (i, i+1).equals(" ")) { //IMPORTANTÍSIMO: COMPARAMOS CON EQUALS
        return entradaTeclado.substring(0, i+1);    }
}
return entradaTeclado; //Caso de que no se detecte ningún espacio devolvemos lo que haya
} //Cierre del método getPrimeraPalabra

public int getLongitud () { //Método de la clase
    return entradaTeclado.length();
} //Cierre del método getLongitud
} //Cierre de la clase

```

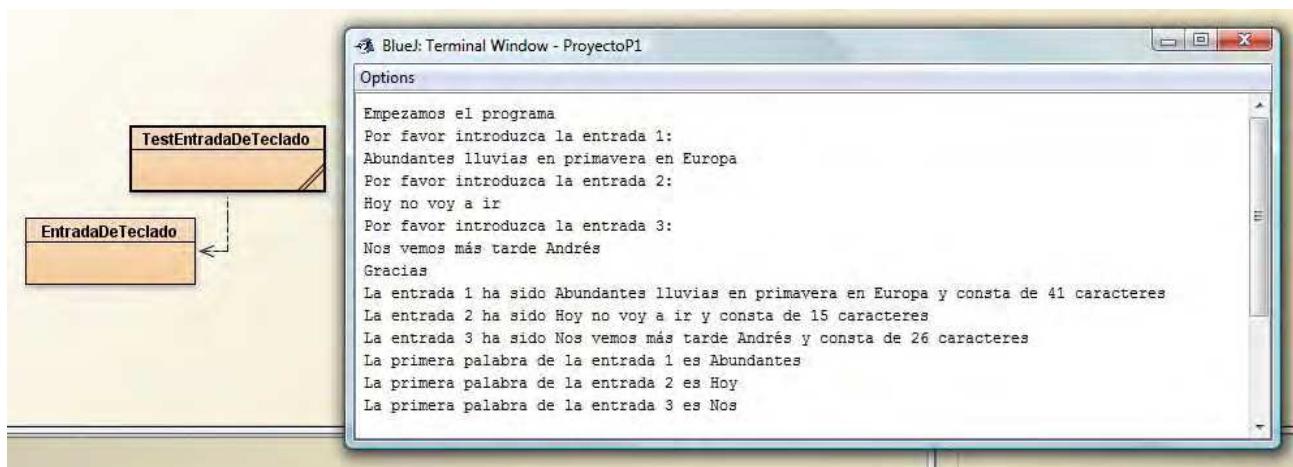
A su vez crea una clase para probar la clase EntradaDeTeclado con este código:

```

// Nuestra clase de prueba aprenderaprogramar.com
public class TestEntradaDeTeclado {
    public static void main (String [ ] args) {
        System.out.println ("Empezamos el programa");
        System.out.println ("Por favor introduzca la entrada 1:");
        EntradaDeTeclado entrada1 = new EntradaDeTeclado();
        System.out.println ("Por favor introduzca la entrada 2:");
        EntradaDeTeclado entrada2 = new EntradaDeTeclado();
        System.out.println ("Por favor introduzca la entrada 3:");
        EntradaDeTeclado entrada3 = new EntradaDeTeclado();
        System.out.println ("Gracias");
        System.out.println ("La entrada 1 ha sido " + entrada1.getEntrada() + " y consta de " + entrada1.getLongitud() + " caracteres");
        System.out.println ("La entrada 2 ha sido " + entrada2.getEntrada() + " y consta de " + entrada2.getLongitud() + " caracteres");
        System.out.println ("La entrada 3 ha sido " + entrada3.getEntrada() + " y consta de " + entrada3.getLongitud() + " caracteres");
        System.out.println ("La primera palabra de la entrada 1 es " + entrada1.getPrimeraPalabra() );
        System.out.println ("La primera palabra de la entrada 2 es " + entrada2.getPrimeraPalabra() );
        System.out.println ("La primera palabra de la entrada 3 es " + entrada3.getPrimeraPalabra() );
    } //Cierre del main
} //Cierre de la clase

```

Ejecuta el programa de test y comprueba cuáles son los resultados. En esta imagen vemos un ejemplo.



Hay varias cuestiones a comentar de nuestro código:

- a) Dentro del constructor hemos realizado una invocación a un método interno de la clase. ¿Por qué? Porque hemos decidido que cada vez que se cree un objeto de la clase se ejecute ese método. En ocasiones un constructor no invoca a métodos, pero en otras ocasiones sí. Depende de lo que decidamos como programadores en función de las circunstancias y necesidades.
- b) En el método `getPrimeraPalabra` hemos escrito `if (entradaTeclado.substring (i, i+1).equals(" ")).` Con esta expresión, dentro de un bucle en el cual vamos recorriendo carácter a carácter el String introducido por el usuario, buscamos determinar dónde se encuentra el primer espacio vacío, que consideramos delimita la primera palabra. Fíjate que estamos usando un método para realizar comparaciones denominado `equals` que es aplicable a los objetos en general y a los String en particular. ¿Por qué no usamos una expresión como `if (entradaTeclado.substring (i, i+1) == " ")`? En primer lugar, porque no funciona. El motivo para ello es que los objetos no se pueden comparar como si se tratara de tipos primitivos. En su momento, dijimos que un **objeto es algo distinto a un tipo primitivo**, aunque “porte” la misma información y que una cadena de caracteres es un objeto. Una peculiaridad de los objetos es que no se pueden comparar usando el operador `==`. En el siguiente epígrafe explicaremos por qué.
- c) En el método `main` usamos expresiones del tipo `EntradaDeTeclado entrada2 = new EntradaDeTeclado();`. Esto es la forma abreviada de escribir:

```
EntradaDeTeclado entrada2;  
entrada2 = new EntradaDeTeclado();
```

Simplemente estamos fusionando la escritura en una línea. A primera vista, parece una forma de escritura un poco redundante. Se repite por todas partes “EntradaDeTeclado”. No obstante, esto tiene su razón de ser como veremos más adelante cuando hablemos de herencia y polimorfismo.

Ejecuta el programa y realiza pruebas con él. Activa la parte de código que está inactiva como comentario y que trata de detectar una letra j usando el operador `==`. Comprueba si funciona o no. Modifica el código para que realice funciones adicionales que se te ocurran. Ten claro que Java no se aprende solo leyendo o estudiando y haciendo ejercicios: has de escribir código y hacer modificaciones y pruebas por tu cuenta para ir familiarizándote con el lenguaje. Debes tratar de diseñar y desarrollar alguna pequeña aplicación para ir aplicando conocimientos adquiridos. Piensa en un pequeño problema y trata de darle solución como programador.

Próxima entrega: CU00662B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

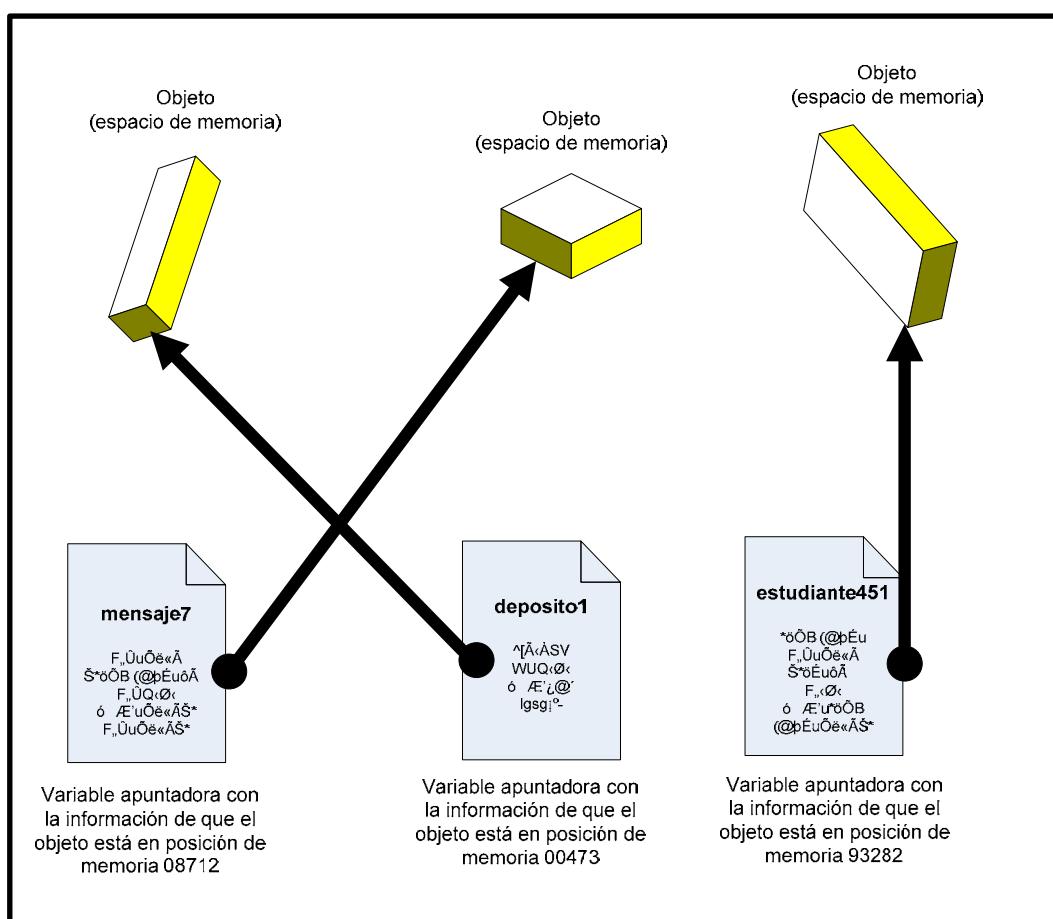
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EL MÉTODO EQUALS EN JAVA. DIFERENCIA ENTRE IGUALDAD E IDENTIDAD DE OBJETOS.

Recordemos ahora algo que hemos comentado anteriormente: **Un objeto es una cosa distinta a un tipo primitivo, aunque “porten” la misma información.** Vamos a tratar de profundizar y aclarar con más detenimiento la diferencia entre objetos y tipos primitivos. Para ello vamos a valernos de un símil: consideremos que un objeto es una caja que puede contener muchas cosas (por ejemplo, un caldero, un bolígrafo, un ventilador, una sábana, etc.).



La variable que referencia al objeto sería una nota de papel donde tenemos escrito dónde se encuentra la caja. Decimos que esta variable contiene una referencia o puntero al objeto.



En este ejemplo vemos cómo cuando usamos una instrucción como `deposito1 = new Deposito();`, en realidad **estamos creando dos cosas**: la variable apuntadora denominada deposito1, que sería como

nuestra nota de papel que nos dice dónde está el objeto, y el objeto o espacio de memoria que queda reservado. Para acceder al objeto hemos de usar la variable apuntadora. La memoria es como un gran almacén con miles de cajas apiladas y si no sabemos dónde está una caja no podremos encontrarla. Cuando creamos un tipo primitivo, creamos una sola cosa, pensemos por ejemplo que creamos un bolígrafo. Y ese bolígrafo llevaría una etiqueta con su nombre (nombre de la variable). Para gestionar tipos primitivos no tenemos la duplicidad entre apuntador y espacio de memoria. Tenemos un espacio de memoria al que podemos acceder directamente.

¿Por qué estas dos formas de gestionar la información? ¿Por qué no almacenar todo de forma que sea accesible directamente? La razón para ello estriba en la eficiencia de computación: todo está pensado para optimizar los rendimientos de los ordenadores y de los programas. Nosotros no tenemos por qué profundizar en estas cuestiones, pero sí conocerlas porque tienen relevancia de cara a la programación.

Consideremos que cuando hacemos una petición a la memoria, esta puede ser de dos tipos:

- Le pedimos un tipo primitivo. Equivaldría a decir directamente “Dame el bolígrafo 783”. Y esta información se nos sirve.
- Le pedimos un objeto. Equivaldría a decir “Dame la caja que está en la posición 93282”. Y esta información se nos sirve.

Supongamos que `mensaje7` es una variable apuntadora a una caja que contiene 2 bolígrafos, y que `estudiante451` es otra variable apuntadora a una caja que contiene otros 2 bolígrafos. Vamos a plantear algunas preguntas:

PREGUNTA	RESPUESTA	RAZONAMIENTO
¿Es idéntico <code>mensaje7</code> que <code>estudiante451</code>? <code>mensaje7 == estudiante451</code>	No, no es igual. False	El contenido de <code>mensaje7</code> es “El objeto está en posición de memoria 08712”, mientras que el contenido de <code>estudiante451</code> es “El objeto está en posición de memoria 93282”. Por tanto no son iguales.
¿Es igual el contenido de <code>mensaje7</code> que el contenido de <code>estudiante451</code>? <code>estudiante7.equals(estudiante451)</code>	Depende	Depende de qué consideremos ser igual. Si consideramos que por contener 2 bolígrafos el contenido es igual la respuesta es sí. Si consideramos que además hemos de comprobar que los bolígrafos sean de la misma marca y color, ahora mismo no sabríamos decir si los contenidos son iguales o no.

Llegamos a conclusiones importantes:

- 1) La comparación usando == no se debe usar para comparar el “contenido” de los objetos, sino únicamente para comparar la información de las variables apuntadoras. Usar == para comparar objetos es un error frecuente en programadores con poca experiencia.
- 2) Para comparar el contenido de los objetos hemos de usar, en general, un método especial del que disponen todos los objetos denominado **equals**. Equals se basa en una definición de igualdad que ha de estar establecida. Por ejemplo, podríamos definir que dos objetos son iguales si contienen el mismo número de bolígrafos, de la misma marca, modelo y color. Para algunos tipos de objetos, como los String, la definición de igualdad que viene incorporada en el API de Java nos va a resultar suficiente. Es decir, *if (miCadena1.equals (miCadena2))* va a funcionar correctamente. Sin embargo, si hemos definido un tipo Deposito y tratamos de utilizar *if (deposito1.equals (deposito2))* no obtendremos un resultado satisfactorio puesto que no hemos definido cuál es el criterio de igualdad.

Planteémonos si es correcta esta sintaxis: *if (entradaTeclado.substring(0,1)=="j")*. Primera cuestión: ¿Qué devuelve el método substring de la clase String? De acuerdo con la documentación del API de Java, devuelve un String (un objeto). Segunda cuestión: ¿Qué es “j”? Al estar entre comillas, se trata de un objeto String. Por tanto estamos preguntando que si un puntero es igual a “j”. Esto no tiene un resultado previsible. Por tanto esta sintaxis no la podemos considerar correcta, ni siquiera aunque aparentemente pueda parecer que funcione. Lo correcto sería usar: *if (entradaTeclado.substring(0,1).equals("j"))*. Aquí lo que estamos haciendo es comparar el contenido de un objeto con el contenido de otro objeto.

Si usamos == comparando dos objetos, en realidad lo que haremos es preguntar si las dos variables que referencian al objeto tienen el mismo puntero, es decir, si ambos objetos son el mismo objeto (identidad).

La comparación de igualdad entre objetos requiere que se defina antes en base a qué se va a considerar que dos objetos son iguales. Por ejemplo, para un objeto String parece bastante claro, será el contenido de la cadena, pero para un objeto Persona quizás no esté tan claro: ¿bastará con que tengan igual DNI o habrá que tener en cuenta nombres y apellidos y otros datos para evitar que un error en el DNI haga que dos personas se consideren iguales? La definición de igualdad se hace con el método equals, que es habitual que se implemente para cualquier clase. Esto lo veremos más adelante. Ahora repasaremos únicamente cómo comparar dos cadenas. La sintaxis será:

Cadena1.equals(Cadena2) ó *("Texto1").equals("Texto2")* devuelve true si la Cadena1 / Texto1 y la Cadena2 / Texto2 tienen el mismo contenido, o false si su contenido es diferente. Por ejemplo: *System.out.println ("Coco y coco son iguales? " + ("Coco").equals("coco"));* devuelve false (cuentan mayúsculas, minúsculas y espacios).

Un método puede aplicarse sobre lo devuelto por otro método. Por ejemplo:

```
System.out.println ("Primera palabra de las dos primeras entradas iguales? " +
entrada1.getPrimeraPalabra().equals(entrada2.getPrimeraPalabra() ));
```

El uso del método equals es exclusivo para objetos. Recordemos que los tipos primitivos no tienen métodos. Por tanto, **los tipos primitivos los compararemos de la forma tradicional usando ==**.

EJERCICIO

Crea una clase en cuyo método main ejecutes una comparación letra a letra usando equals de dos palabras usando bucles. Por ejemplo si las palabras son “avispa” y “ave” el programa debe dar como resultado: ¿Letra 1 igual en las dos palabras? True. ¿Letra 2 igual en las dos palabras? True ¿Letra 3 igual en las dos palabras? False ¿Letra 4 igual en las dos palabras? La palabra 2 no tiene letra 4 ¿Letra 5 igual en las dos palabras? La palabra 2 no tiene letra 5 ¿Letra 6 igual en las dos palabras? La palabra 2 no tiene letra 6.

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00663B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

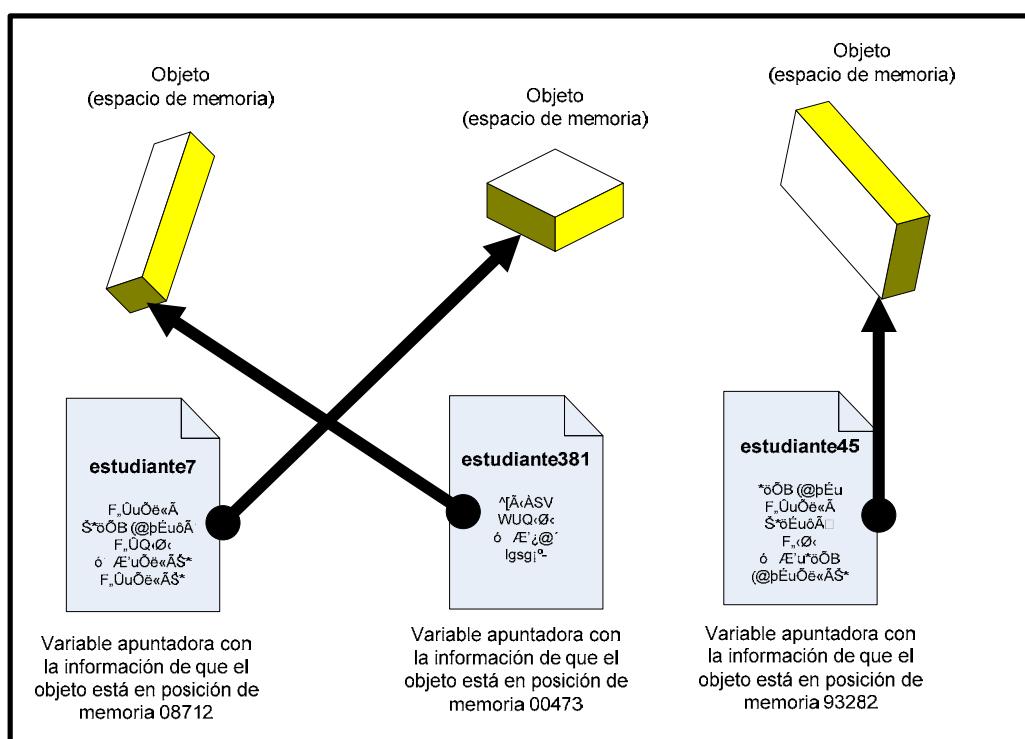
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

ASIGNACIÓN DE IGUALDAD CON TIPOS PRIMITIVOS Y OBJETOS.

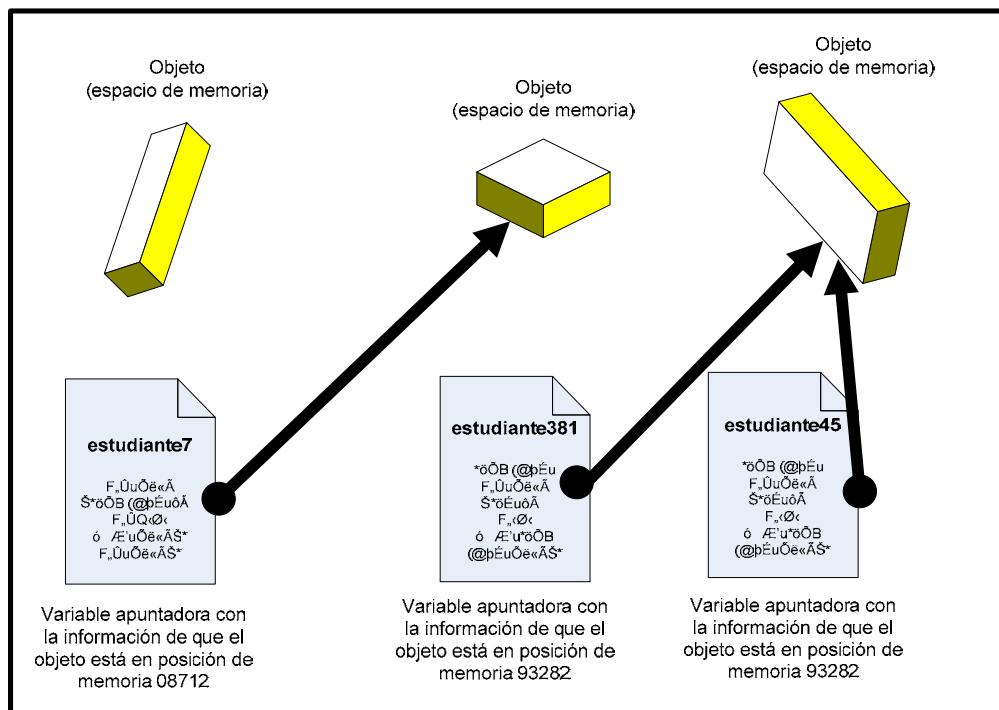
Si como hemos explicado anteriormente los tipos objeto no se almacenan directamente sino mediante referencias al objeto, ¿Qué efecto tiene hacer una asignación de tipo = entre objetos? En primer lugar, hay que tener en cuenta que para establecer asignaciones de este tipo entre objetos debe haber coincidencia de tipos. Es decir, tratar de establecer `estudiante478 = deposito3`; nos llevará a un mensaje de error del compilador de tipo “*Incompatible types – found Deposito but expected Estudiante*”.



Consideremos que hubiéramos definido un tipo `Estudiante` y que hemos creado tres objetos referenciados por las variables: `estudiante7`, `estudiante381`, `estudiante45`. Gráficamente podríamos representarlo así:



Ahora supongamos que establecemos: estudiante381 = estudiante45. El significado de esta instrucción es que la variable apuntadora de estudiante381 se convierte en idéntica a la variable apuntadora estudiante45. Es decir, ahora ambas variables referencian al mismo objeto. Gráficamente:



Si nos fijamos en este último gráfico, comprobaremos una cosa: dos variables tienen la misma información (apuntan al mismo objeto) y un objeto ha dejado de estar apuntado, se ha quedado "flotando". ¿Cómo podemos acceder a ese objeto? La respuesta es que ya no podemos acceder. Este tipo de situaciones en principio no son deseables, ya que supone dejar "información" a la que no podemos acceder y esto no puede considerarse adecuado o razonable. Al hacer una asignación como `estudiante381 = estudiante45` lo que hacemos es tener dos variables referenciadoras con la misma información, lo que nos lleva a que solo podamos acceder a un único objeto. Después de hecha la asignación ambas variables nos referencian al mismo objeto, podríamos decir que son redundantes: dos nombres para la misma cosa. Dos variables que apuntan al mismo objeto se dice que son "alias" de ese objeto.

¿Qué ocurre con los tipos primitivos? En los tipos primitivos, al no tener la configuración apuntador <-> contenido, podemos usar las asignaciones = sin ningún problema. Por ejemplo: `int i = 3; int j = 5; i = j;` supone que la variable i toma el valor de la variable j, es decir, i pasa a valer 5.

Dentro de los objetos tenemos el caso que ya hemos citado en algunas ocasiones que es un tanto especial de los tipo String. En este caso, sí es posible usar el operador de asignación = para transferir el contenido de unas variables a otras porque como ya hemos indicado, se trata de objetos con un comportamiento un tanto singular.

Volvamos a los tipos objeto. Si no podemos usar el operador = ¿Cómo podemos asignar a un objeto el mismo contenido que otro objeto pero manteniendo que ambas variables referencien a objetos independientes? Vamos a citar, sin entrar en profundidad, algunas maneras:

- 1) Partimos de un primer objeto con un contenido. Creamos otro segundo objeto usando la instrucción `new`. Seguidamente, establecemos el contenido de campos del segundo objeto usando los métodos `set` y asignando como contenidos los valores devueltos por los métodos `get` del primer objeto. Por ejemplo: `estudiante381.setNombre (estudiante45.getNombre());`.
- 2) Buscamos formas para clonar objetos en el API de Java. Una posible vía para crear una copia de un objeto en otro objeto sería usar el método `clone`. No vamos a explicar su uso ahora.
- 3) Definimos un constructor que nos permita pasar como parámetro un objeto y cuyo código tenga las instrucciones precisas para que el objeto que se cree tenga el mismo contenido que el objeto que se pasa como parámetro. La creación del objeto con el mismo contenido la lograríamos invocando ese constructor, por ejemplo: `estudiante 381 = new Estudiante (estudiante45);`
- 4) Otras vías.

A modo de conclusión hay que retener algo importante: no podemos usar el operador de asignación `=` entre objetos de la misma forma en que lo hacemos para tipos primitivos.

REPASO Y EJEMPLOS SOBRE IGUALDAD, IDENTIDAD Y MÉTODO EQUALS

Consideremos que un objeto ocupa un espacio de memoria. Ahora diremos que hay una o varias variables que apuntan a ese objeto. Supongamos la clase Persona y dos objetos creados como:

```
Persona p1 = new Persona("José Ramírez Mota", 32);
Persona p2 = new Persona("José Ramírez Mota", 32);
```

Ahora nos podemos plantear lo siguiente: `p1 == p2`? Esta pregunta en Java se traduce así: ¿Es el objeto al que apunta `p1` igual al objeto al que apunta `p2`? Es decir, ¿son el mismo objeto los objetos apuntados por `p1` y `p2`, y por tanto existe una relación de identidad entre ambos? La respuesta es que no, no son el mismo objeto, pues cada vez que usamos el constructor, cada vez que usamos `new`, creamos un nuevo objeto. Seguramente no queríamos preguntarnos si las variables apuntadoras apuntaban al mismo objeto, sino si los objetos eran "iguales". Ahora bien, la igualdad entre tipos primitivos sí es evaluable con relativa facilidad porque no distinguimos entre identidad e igualdad, en tipos primitivos ambas cosas podemos decir que son lo mismo: p.ej. ¿Es `32 == 21` ó el carácter '`f`' =='`F`'? Obviamente no. Pero **la igualdad entre objetos en Java se basa en que exista una definición previa de cuál es el criterio de igualdad**, y esta definición es la que dé el método `equals` aplicable al objeto (todo objeto en principio tendrá un método `equals`). ¿Qué ocurre si hacemos `p1 = p2`? En este caso un objeto deja de estar apuntado por variable alguna. Las dos variables pasan a apuntar al mismo objeto y por tanto `p1 == p2` devuelve `true`.

Recordar que los String, Integer, etc. son objetos y que por tanto no se pueden comparar usando `==`. Consideremos este caso:

```
Persona p1 = new Persona("Andrés Hernández Suárez", 32, "54221893-D", "Economista");
Persona p2 = new Persona("Andrés Hernández Suárez", 32, "54221893-D", "Abogado");
```

¿Cómo establecemos si dos personas son iguales? El concepto de igualdad entre objetos, en Java, tiene que ser definido para que el compilador sepa a qué atenerse si se comparan dos objetos, en este caso a dos personas. La comparación usará la sintaxis:

```
if (objeto1.equals(objeto2)) { Instrucciones si se cumple el criterio de igualdad..... } else { Instrucciones si no se cumple el criterio de igualdad }
```

Por defecto, Java permite el uso del método equals para todo objeto, pero el criterio por defecto de Java no nos va a ser útil para clases creadas por nosotros: tendremos que definir nuestro criterio de igualdad dentro de nuestra clase. Veremos más adelante cómo se define un criterio de igualdad especificado por el programador para poder comparar objetos.

EJERCICIO

Considera una clase Java que se denomina TripulacionAvion y que tiene como atributos a tres objetos de tipo Persona: Persona piloto; Persona copiloto1; Persona copiloto2.

- a) ¿Sería posible que al crear un objeto TripulacionAvion se produjera que piloto, copiloto1 y copiloto2 apuntaran a un mismo objeto, es decir, que existiera una relación de identidad entre los tres atributos?
- b) ¿Existiría relación de identidad cuando creamos un objeto TripulacionAvion entre los tres atributos si no se inicializaran en el constructor?
- c) ¿Cuál sería el contenido de los atributos si no se inicializan en el constructor y creamos un objeto de tipo TripulacionAvion?

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00664B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

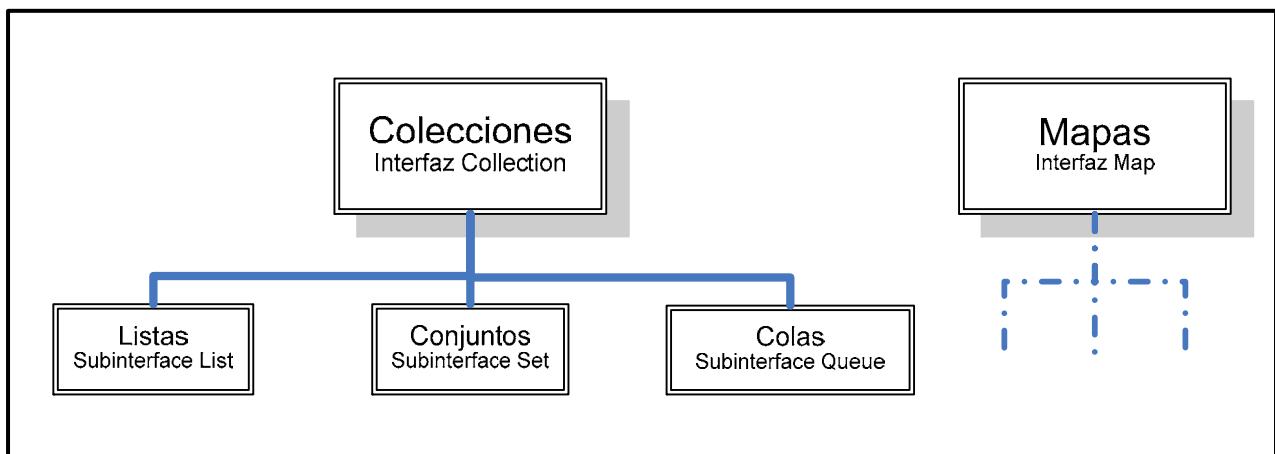
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

COLECCIONES DE OBJETOS DE TAMAÑO VARIABLE. CONTENEDORES.

Una colección de objetos es un objeto que puede almacenar un número variable de elementos siendo cada elemento otro objeto. Siguiendo con nuestro símil, podríamos ver una colección como una caja que contiene más cajas dentro. Puede haber distintos tipos de colecciones de tamaño “flexible”, es decir, que se pueden encoger o agrandar según las necesidades.



En Java se distinguen diversos tipos, agrupados en una estructura general similar a la que muestra este esquema.



Por el momento consideraremos que una interfaz viene siendo algo así como una protoclase: define ciertas cosas que van a compartir cierto número de subinterfaces y de clases.

Cada tipo de colección tiene unas características.

- Una lista (list)** es una colección de objetos donde cada uno de ellos lleva un índice asociado. Así, podríamos tener una lista con los nombres de las personas que han utilizado un servicio de acceso a internet que podría ser: usuarios --> (Juan R.R., Sara G.B., Rodolfo M.N., Pedro S.T., Claudio R.S., Juan R.R.). Donde cada contenido va asociado a un índice, usuario(0) sería Juan R.R., usuario(1) sería Sara G.B., usuario(2) sería Rodolfo M.N. y así sucesivamente. En una lista podemos insertar y eliminar objetos de posiciones intermedias. Ejemplos de listas son la clase *ArrayList* y *LinkedList* del API de Java.

- b) **Un conjunto (set)** sería una colección de objetos que no admite duplicados. Siguiendo el ejemplo anterior, un conjunto nos serviría para saber los usuarios distintos que han utilizado el servicio de acceso a internet, pero no tendríamos información sobre el orden y una misma persona no aparecería más de una vez, ni siquiera aunque hubiera utilizado el servicio varias veces. Ejemplo de conjunto sería la clase *HashSet* del API de Java.
- c) **Una cola (queue)** sería una colección de objetos que se comportan como lo haría un grupo de personas en la cola de una caja de un supermercado. Los objetos se van poniendo en cola y el primero en salir es el primero que llegó.
- d) **Una pila (stack)** sería una colección de objetos que se comportan como lo haría un montón de objetos apilados, el último en añadirse queda en la cima de la pila y el primero en salir es el último que ha llegado. El primero en llegar está en el fondo de la pila.
- e) **Existen otros tipos de colecciones.**

Lo cierto es que las estructuras de datos en Java ofrecen muchas posibilidades y variantes. Por ejemplo, podemos tener conjuntos sin orden entre los elementos, pero también conjuntos ordenados. Y podemos tener colas donde el objeto que sale primero no es el primero que llegó, sino el que tiene mayor prioridad o “urgencia” por salir. Para escoger un tipo de comportamiento u otro existen distintas clases que se catalogan como colecciones en el API de Java. Incluso existen clases que sirven para agrupar objetos que Java no clasifica como colecciones, aunque su funcionamiento es muy similar. Sería el caso de los Maps, objetos que contienen parejas de objetos donde un elemento sirve para encontrar al otro, al igual que un listín telefónico contiene parejas “Nombre de personas – Datos de dirección y teléfono” y el nombre de persona es lo que nos sirve para encontrar sus datos asociados. En la nomenclatura de Java, los Maps no son colecciones. Por eso a veces se usa el término “contenedores de objetos” para hacer referencia a listas, conjuntos, colas, mapas, etc. que al fin y al cabo son objetos que contienen más objetos (como una caja que contiene más cajas).

¿Cómo saber qué clase elegir? Hay varios factores a tener en cuenta, entre ellos el número de datos que tenemos que gestionar (no es lo mismo trabajar con una colección de 50 objetos que con una colección de 50.000 objetos) y el tipo de procesos que tenemos que realizar con ellos (no es lo mismo una lista en que los nuevos elementos se añaden casi siempre al final de la lista que una lista donde los nuevos elementos se añaden frecuentemente en posiciones intermedias). Cada clase resulta más eficiente que otra para realizar determinados procesos. Esto es de especial interés cuando hay que gestionar muchos datos. Si hablamos de sólo unas decenas de datos no vamos a ser capaces de apreciar diferencias de rendimientos.

Nosotros no vamos a estudiar todas las clases contenedoras de objetos porque excedería los objetivos de este curso. Citaremos simplemente algunas clases que son frecuentemente usadas por los programadores porque sirven para cubrir muchas necesidades de programación. Estas clases son *ArrayList*, *LinkedList*, *HashSet* y *HashMap*. Vamos a centrarnos primeramente en una de las clases más usadas, la clase *ArrayList* del API de Java. Estudiando esta clase aprenderemos cosas que después nos serán útiles para aplicar a cualquier contenedor de objetos. No obstante, siempre habremos de consultar la documentación del API de Java para conocer a fondo una clase.

EJERCICIO

Consulta la clase Stack en el API de Java. ¿Cómo se llama el método para consultar el objeto en la cima de la pila? ¿Cómo se llama el método para consultar si la pila está vacía? ¿El método pop podríamos clasificarlo como tipo procedimiento, como tipo función, o como tipo mixto (procedimiento y función)?

Próxima entrega: CU00665B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

LA CLASE ARRAYLIST DEL API DE JAVA. LISTAS REDIMENSIONABLES.

La clase `ArrayList` podríamos encuadrarla de la siguiente manera: Colecciones --> Listas --> `ArrayList`. Esta clase pertenece a la biblioteca `java.util`. Por tanto, para emplearla en nuestras clases o programas escribiremos como código en cabecera `import java.util.ArrayList` (o de forma más genérica `import java.util.*`). **Un objeto `ArrayList` será una lista redimensionable** en la que tendremos disponibles los métodos más habituales para operar con listas.



Como métodos para operar con listas podemos señalar: añadir un objeto en una posición determinada, añadir un objeto al final de la lista, recuperar un objeto situado en determinada posición, etc. Los objetos de un `ArrayList` tienen un orden, que es el orden en el que se insertaron en la lista.

Un aspecto a tener muy presente: hablamos de colecciones de objetos. Por tanto, un `ArrayList` no puede ser una lista de enteros como tipo primitivo (`int`) pero sí de objetos `Integer`. La sintaxis que emplearemos con `ArrayList` es la siguiente:

(Declaración del objeto `ArrayList`) : `private ArrayList<TipoDeObjetosEnLaColección> NombreDelArrayList;`

(Creación de un objeto): `NombreDeObjeto = new ArrayList<TipodeObjetosEnLaColección>();`

(Uso del método reemplazar objeto existente): `NombreDelArrayList.set (int índice, E elemento);`

(Uso del método añadir al final): `NombreDelArrayList.add (objeto_a_añadir);`

(Uso del método obtener el número de objetos en la lista): `NombreDelArrayList.size();`

(Uso del método extraer un objeto de cierta posición): `NombreDelArrayList.get (posición);`

Si consultas la documentación de la clase, verás que la clase `ArrayList` tiene varios constructores. En este caso estamos utilizando el constructor sin parámetros que crea una lista `ArrayList` vacía con una capacidad inicial para diez objetos (la capacidad es modificable luego y se amplía automáticamente a medida que vamos añadiendo elementos). Escribe y compila este código para probar el funcionamiento de `ArrayList`.

```
//Ejemplo de uso ArrayList aprenderaprogramar.com
import java.util.ArrayList; //Los import deben ir siempre al principio antes de declarar la clase

//Esta clase representa una lista de nombres manejada con la clase ArrayList de Java
public class ListaNombres {
    private String nombreDeLaLista; //Establecemos un atributo nombre de la lista
    private ArrayList<String> listadenombres; //Declaramos un ArrayList que contiene objetos String

    public ListaNombres (String nombre) { //Constructor: crea una lista de nombres vacía
        nombreDeLaLista = nombre;
        listadenombres = new ArrayList<String>(); //Creamos el objeto de tipo ArrayList
    } //Cierre del constructor
```

```

public void addNombre (String valor_nombre) { listadenombres.add (valor_nombre); } //Cierre del método

public String getNombre (int posicion) { //Método
    if (posicion >= 0 && posicion < listadenombres.size() ) {
        return listadenombres.get(posicion);
    }
    else { return "No existe nombre para la posición solicitada";}
} //Cierre del método

public int getTamaño () { return listadenombres.size(); } //Cierre del método

public void removeNombre (int posicion) { //Método
    if (posicion >= 0 && posicion < listadenombres.size() ) {
        listadenombres.remove(posicion);
    }
    else { } //else vacío. No existe nombre para la posición solicitada, no se ejecuta ninguna instrucción
} //Cierre del método removeNombre

} //Cierre de la clase

```

Crea un objeto y añade varios nombres en el ArrayList. Prueba el correcto funcionamiento de los métodos disponibles. Ten en cuenta que al eliminar un objeto de la colección, todos los elementos posteriores se renumeran disminuyendo su índice una posición, automáticamente. Muchas colecciones, entre ellas ArrayList, tienen una numeración implícita de cada uno de los objetos de los que consta. Esta numeración va desde cero hasta (número de elementos -1), es decir, en una colección de 18 personas los índices van desde persona(0) hasta persona(17), o en una colección que tiene 155 depósitos, los índices irán de deposito(0) a deposito(154). El número de elementos en la colección lo podemos obtener en cualquier momento utilizando el método size(). Si intentamos acceder a una colección con un índice no válido obtendremos un error del tipo “IndexOutOfBoundsException” (desbordamiento).

En apartados anteriores definimos una clase denominada EntradaDeTeclado que servía para recibir datos de entrada introducidos por el usuario mediante el teclado. Vamos ahora a utilizar esa clase para crear un pequeño test de la clase ListaNombres. Escribe el siguiente código:

```

// Aquí el test con el método main ejemplo aprenderaprogramar.com
public class TestListaNombres {

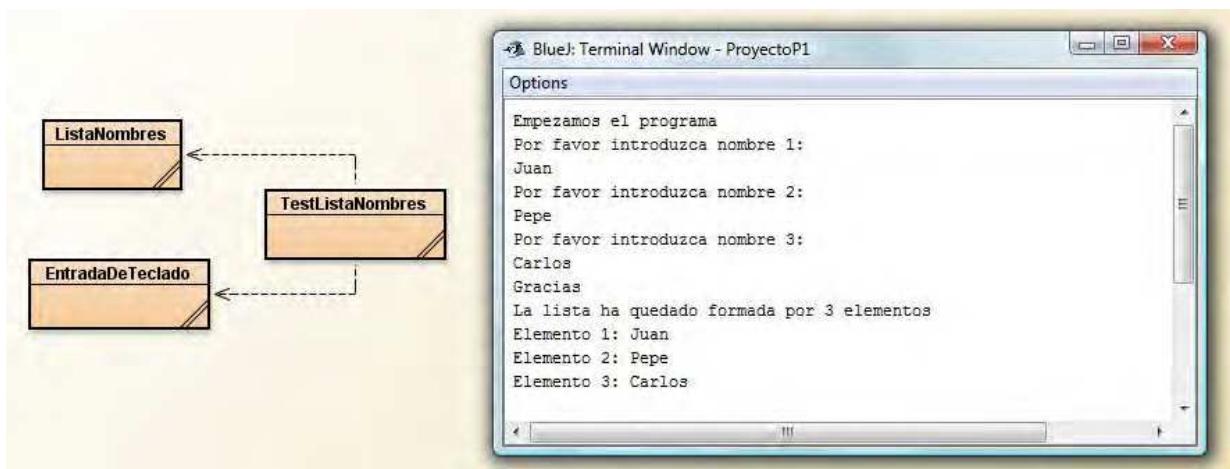
    public static void main (String [ ] args) {

        System.out.println ("Empezamos el programa");
        System.out.println ("Por favor introduzca nombre 1:"); EntradaDeTeclado entrada1 = new EntradaDeTeclado();
        System.out.println ("Por favor introduzca nombre 2:"); EntradaDeTeclado entrada2 = new EntradaDeTeclado();
        System.out.println ("Por favor introduzca nombre 3:"); EntradaDeTeclado entrada3 = new EntradaDeTeclado();
        System.out.println ("Gracias");
        ListaNombres lista1 = new ListaNombres("Nombres introducidos por usuario");
        lista1.addNombre (entrada1.getEntrada() ); lista1.addNombre (entrada2.getEntrada() );
        lista1.addNombre (entrada3.getEntrada() );
        System.out.println ("La lista ha quedado formada por " + lista1.getTamaño() + " elementos");
        System.out.println ("Elemento 1: " + lista1.getNombre(0) );
        System.out.println ("Elemento 2: " + lista1.getNombre(1) );
        System.out.println ("Elemento 3: " + lista1.getNombre(2) );
    } //Cierre del main

} //Cierre de la clase

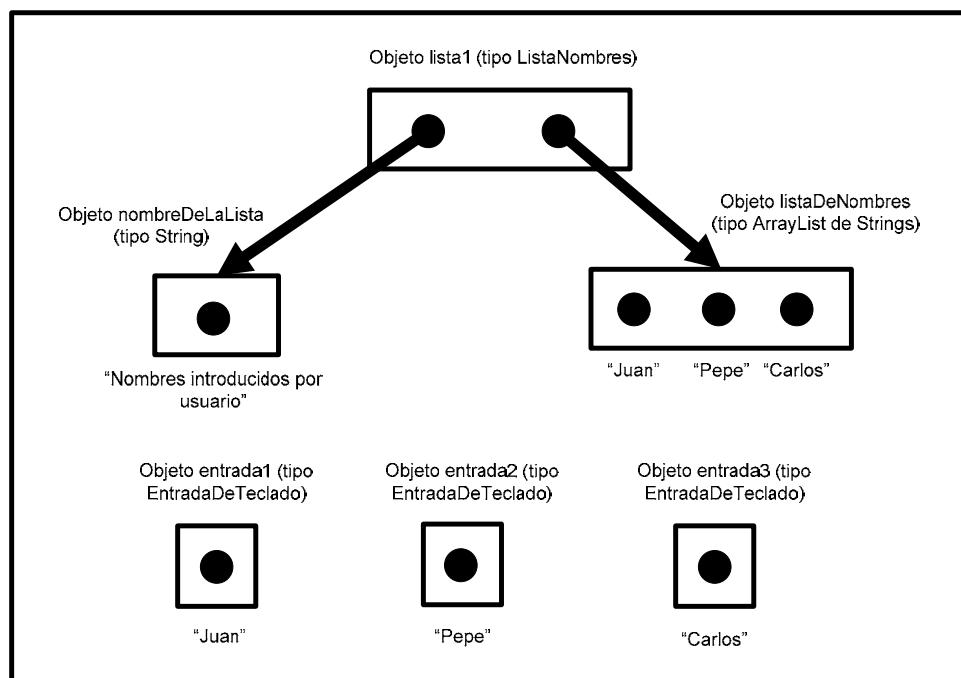
```

El diagrama de clases y el resultado de ejecutar el programa son de este tipo:



La clase `TestListaNombres` usa tanto la clase `ListaNombres` como la clase `EntradaDeTeclado`. El código de `EntradaDeTeclado` ya lo teníamos escrito por haberlo usado en un programa anterior y por tanto no hemos de escribirlo de nuevo.

El siguiente diagrama muestra los objetos que intervienen en el programa anterior:



Vamos a interpretar este diagrama. El objeto `lista1` contiene dos objetos: un objeto de tipo `String` y un objeto de tipo `ArrayList`. A su vez el `ArrayList` contiene tres objetos de tipo `String`. Por otro lado tenemos los objetos `EntradaDeTeclado`, cada uno de los cuales contiene un objeto de tipo `String`. ¿Por qué no salen flechas desde los objetos `EntradaDeTeclado`? Porque la flecha indica hacia dónde apunta una variable referenciadora de un objeto. Las variables referenciadoras de los objetos `EntradaDeTeclado` no apuntan (no contienen) otros objetos. Simplemente nos sirven para extraer algo. Fíjate que usamos `lista1.addNombre (entrada1.getEntrada())`; en vez de `lista1.addNombre (entrada1)`. La diferencia radica en que en el primer caso añadimos un `String` al `ArrayList`, mientras que en el segundo caso estaríamos añadiendo un objeto `EntradaDeTeclado`.

CONCEPTO DE CLASE GENÉRICA O CLASE PARAMETRIZADA EN JAVA

Fijémonos en la sintaxis de estas declaraciones:

DECLARACIÓN EN OTRAS CLASES	DECLARACIÓN DE TIPO ARRAYLIST
<pre>private String cadenaDeTexto; private int numeroDepositosGrupo; private Persona jefeDePersonal;</pre>	<pre>private ArrayList<String> listaDeNombres;</pre>

Mientras que para declarar un String o una Persona no necesitamos nada más que el nombre de la clase, para declarar un ArrayList hemos de usar además un parámetro especificado entre los símbolos < y >. Por ejemplo:

```
private ArrayList<String> listaDeFrutas; //Lista que admite solo Strings
private ArrayList<Persona> miembrosDelClub; //Lista que admite solo Personas
private ArrayList<Deposito> grupoDeDepositos; //Lista que admite solo Depósitos
```

Solo podemos añadir objetos del tipo declarado al parametrizar la clase. Este tipo de clases, que requieren un tipo como parámetro, se denominan “**clases genéricas o parametrizadas**”. Dado que ArrayList es una clase que puede definir una lista de distintos tipos de objeto (como Strings, Personas, Depósitos...) decimos que la clase ArrayList define potencialmente muchos tipos puesto que con ella podemos crear listas de cualquier tipo de objetos. Un tipo ArrayList<String> es distinto a un tipo ArrayList<Deposito>, por tanto podemos tener infinitas clases basadas en la clase genérica ArrayList.

En la documentación del API de Java, que una clase está parametrizada se refleja en su documentación. Por ejemplo, si consultamos la documentación de ArrayList veremos que en cabecera aparece como Class ArrayList<E> lo que nos indica que se requiere un parámetro para crear objetos de tipo ArrayList. Si consultamos la documentación de la clase HashMap comprobaremos que en cabecera aparece como Class HashMap <K, V> lo cual nos indica que se requieren dos parámetros para crear objetos de tipo HashMap.

EJERCICIO

Crea una clase denominada ListaCantantesFamosos que al ser inicializada contenga un ArrayList con tres Strings que sean el nombre de cantantes famosos. Crea una clase test con el método main que inicialice un objeto ListaCantantesFamosos, pida dos cantantes famosos más al usuario, los añada a la lista y muestre el contenido de la lista por pantalla. Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00666B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EL FOR EXTENDIDO O BUCLES FOR EACH EN JAVA.

A partir de Java 5 se introdujo una nueva forma de uso del for, a la que se denomina “for extendido” o “for each”. Esta forma de uso del for, que ya existía en otros lenguajes, facilita el recorrido de objetos existentes en una colección sin necesidad de definir el número de elementos a recorrer. La sintaxis que se emplea es:



```
for ( TipoARecorrer nombreVariableTemporal : nombreDeLaColección ) {
    Instrucciones
}
```

Fíjate que en ningún momento se usa la palabra clave *each* que se usa en otros lenguajes, aunque al for muchas veces se le nombra como *for each*. Para saber si un for es un for extendido o un for normal hemos de fijarnos en la sintaxis que se emplea. La interpretación que podemos hacer de la sintaxis del for extendido es: “Para cada elemento del tipo TipoARecorrer que se encuentre dentro de la colección nombreDeLaColección ejecuta las instrucciones que se indican”. La variable local-temporal del ciclo almacena en cada paso el objeto que se visita y sólo existe durante la ejecución del ciclo y desaparece después. Debe ser del mismo tipo que los elementos a recorrer. Ejemplo:

```
//Ejemplo aprenderaprogramar.com
public void listarTodosLosNombres () {
    for (String i: listaDeNombres) {
        System.out.println (i); //Muestra cada uno de los nombres dentro de listaDeNombres
    }
}
```

En este tipo de ciclos podemos darle un nombre más descriptivo a la variable temporal, por ejemplo:

```
//Ejemplo aprenderaprogramar.com
public void listarTodosLosNombres () {
    for (String nombre: listaDeNombres) {
        System.out.println (nombre);
    }
}
```

Un ejemplo de llamada desde un método main (u otro lugar) sería:

```
//Ejemplo aprenderaprogramar.com
System.out.println ("Mostramos todos los nombres con un ciclo for-each");
lista1.listarTodosLosNombres();
```

El for extendido tiene algunas ventajas y algunos inconvenientes. No se debe usar siempre. Su uso no es obligatorio, de hecho, como hemos indicado, en versiones anteriores ni siquiera existía en el lenguaje. En vez de un for extendido podemos preferir usar un ciclo while. Lo haríamos así:

```
//Ejemplo aprenderaprogramar.com
int i = 0;
while (i < lista1.size() ) {    System.out.println (lista1.getNombre(i) );
                                i++; }
```

El ciclo for-each es una herramienta muy útil cuando tenemos que realizar recorridos completos de colecciones, por lo que lo usaremos en numerosas ocasiones antes que ciclos for o while que nos obligan a estar pendientes de más cuestiones (por ejemplo en este caso con el while, de llevar un contador, llamar en cada iteración a un método, etc.). Un for extendido en principio recorre todos y cada uno de los elementos de una colección. Sin embargo, podemos introducir un condicional asociado a una sentencia break; que aborte el recorrido una vez se cumpla una determinada condición. Escribe y compila el siguiente código ejemplo de uso de un for extendido:

```
import java.util.ArrayList;
//Test del for extendido ejemplo aprenderaprogramar.com
public class TestForExtendido {
    public static void main (String [ ] Args) {
        ArrayList <String> jugadoresDeBaloncesto = new ArrayList<String> ();
        jugadoresDeBaloncesto.add ("Michael Jordan"); jugadoresDeBaloncesto.add ("Kobe Bryant");
        jugadoresDeBaloncesto.add ("Pau Gasol"); jugadoresDeBaloncesto.add ("Drazen Petrovic");
        int i = 0;
        System.out.println ("Los jugadores de baloncesto en la lista son: ");
        for (String nombre : jugadoresDeBaloncesto) { System.out.println ((i+1) + ".- " +nombre);
                                                        i++; }
    } } //Cierre del main y de la clase
```

EJERCICIO

Crea una clase denominada ListaCantantesFamosos que al ser inicializada contenga un ArrayList con tres Strings que sean el nombre de cantantes famosos. Crea una clase test con el método main que inicialice un objeto ListaCantantesFamosos y usando un for extendido muestre los cantantes en la lista por pantalla. Se debe pedir al usuario un nombre más de cantante famoso, y una vez introducido mostrar la lista actualizada usando un for extendido. Una vez mostrada la lista actualizada, se debe dar opción a elegir entre volver a introducir otro cantante o salir del programa (se podrán introducir tantos cantantes como se desee, para ello usa un bucle while que dé opción a elegir al usuario). Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00667B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

TIPO ITERATOR Y MÉTODO ITERATOR EN JAVA

El uso del bucle for-each tiene algunos inconvenientes. Uno de ellos, que para recorrer la colección nos basamos en la propia colección y por tanto no podemos (o al menos no debemos) manipularla durante su recorrido. Supongamos que vamos recorriendo una lista de 20 objetos y que durante el recorrido borramos 5 de ellos. Probablemente nos saltará un error porque Java no sabrá qué hacer ante esta modificación concurrente.



Sería como seguir un camino marcado sobre unas baldosas y que durante el camino nos movieran las baldosas de sitio: no podríamos seguir el camino previsto. Este tipo de problemas se suelen presentar con un mensaje de error del tipo `java.util.ConcurrentModificationException`.

El uso de operaciones sobre colecciones usando un for tradicional también puede dar lugar a resultados no deseados. En general, operar sobre una colección al mismo tiempo que la recorremos puede ser problemático. Este problema queda salvado mediante un recurso del API de Java: los objetos tipo Iterator. **Un objeto de tipo Iterator funciona a modo de copia para recorrer una colección**, es decir, el recorrido no se basa en la colección “real” sino en una copia. De este modo, al mismo tiempo que hacemos el recorrido (sustentado en la copia) podemos manipular la colección real, añadiendo, eliminando o modificando elementos de la misma. Para poder usar objetos de tipo Iterator hemos de declarar en cabecera `import java.util.Iterator;`. La sintaxis es la siguiente:

```
Iterator <TipoARecorrer> it = nombreDeLaColección.iterator();
```

Esta sintaxis resulta un tanto confusa. Por ello vamos a tratar de analizarla con detenimiento. En primer lugar, declaramos un objeto de tipo Iterator. La sentencia para ello es `Iterator <TipoARecorrer> it;` donde `it` es el nombre del objeto que estamos declarando. El siguiente paso es inicializar la variable para que efectivamente pase a contener un objeto. Pero Iterator carece de constructor, así que no podemos usar una sentencia de tipo `new Iterator<TipoARecorrer>();` porque no resulta válida. Iterator es una construcción un tanto especial. De momento nos quedamos con la idea de que es algo parecido a una clase. Dado que no disponemos de constructor, usamos un método del que disponen todas las colecciones denominado `iterator()` y que cuando es invocado devuelve un objeto de tipo Iterator con una copia de la colección. Por tanto hemos de distinguir:

- a) Iterator con mayúsculas, que define un tipo.
- b) El método iterator (con minúsculas) que está disponible para todas las colecciones y que cuando es invocado devuelve un objeto de tipo Iterator con una copia de la colección.

¿Cómo sabemos cuándo nos referimos al tipo Iterator y cuándo al método iterator()? Prestando atención a las mayúsculas/minúsculas y al contexto de uso. Si el nombre de una colección va seguido de `.iterator()` sabremos que estamos utilizando un método para obtener el iterador de la colección. Por las

minúsculas, por los paréntesis, y por ir a continuación del nombre de la colección. Iterator es un tipo genérico o parametrizado, porque requiere que definamos un tipo complementario cuando declaramos objetos de tipo Iterator. Los objetos de tipo Iterator tienen como métodos:

hasNext(): devuelve true si el objeto no es el último de la colección (the object has next).

next(): hace que el objeto pase a referenciar al siguiente elemento de la lista.

remove(): elimina de la colección que se está recorriendo el último objeto devuelto por next().

Uno de los métodos disponibles para la clase String es *contains*. La sintaxis para su uso es “Cadena”.contains (“textoabutar”), que devuelve true si *Cadena* contiene *textoabutar*. Consideremos el siguiente ejemplo:

```
//Ejemplo aprenderaprogramar.com
if (colección.next().contains(cadena) ) { System.out.println ("Cadena encontrada en " + colección.next()); }
```

Este código presenta un problema. ¿Cuál es? Que con la primera referencia a colección.next() se evalúa si un ítem de la colección contiene la variable *cadena*, y cuando le decimos que imprima que se ha encontrado la cadena en colección.next() no nos devuelve el ítem deseado. ¿Por qué? Porque cada vez que aparezca el método next() se devuelve el siguiente objeto dentro de la colección.

Por eso, en el siguiente ejemplo, donde queremos evaluar y mostrar el String que se analiza, utilizamos otro String temporal para almacenar el que nos devuelve cada llamada al método next(). Escribe y compila este código:

```
import java.util.Iterator;
import java.util.ArrayList;

public class TestUsIterator { //Ejemplo uso iterator aprenderaprogramar.com

    public static void main (String [ ] Args) {
        ArrayList <String> listaDeNombres = new ArrayList <String> ();
        listaDeNombres.add("Juan Pérez Sánchez");
        listaDeNombres.add("José Alberto Reverón Montes");
        String cadenaBuscar = "Alberto";
        System.out.println ("La cadena que buscamos es " + cadenaBuscar);

        Iterator<String> it = listaDeNombres.iterator(); //Creamos el objeto it de tipo Iterator con String
        String tmpAnalizando;
        while ( it.hasNext() ) { //Utilizamos el método hasNext de los objetos tipo Iterator
            tmpAnalizando = it.next(); //Utilizamos el método next de los objetos tipo Iterator
            System.out.println ("Analizando elemento... " + tmpAnalizando);
            if (tmpAnalizando.contains(cadenaBuscar) ) {
                System.out.println ("Cadena encontrada!!!!");

            } else {} //else vacío. No hay acciones a ejecutar.
        } //Cierre del while
    } //Cierre del main
} //Cierre de la clase
```

Además de las ventajas propias de trabajar con una copia en vez de con la colección original, otro aspecto de interés de la clase Iterator y el método iterator radica en que no todas las colecciones de objetos en Java tienen un índice entero asociado a cada objeto, y por tanto no se pueden recorrer basándonos en un índice. En cambio, siempre se podrán recorrer usando un iterador.

EJERCICIO

Crea una clase denominada ListaCantantesFamosos que disponga de un atributo ArrayList listaCantantesFamosos que contenga objetos de tipo CantanteFamoso. La clase debe tener un método que permita añadir objetos de tipo CantanteFamoso a la lista. Un objeto de tipo CantanteFamoso tendrá como atributos nombre (String) y discoConMasVentas (String), y los métodos para obtener y establecer los atributos. Crea una clase test con el método main que inicialice un objeto ListaCantantesFamosos y añade manualmente dos objetos de tipo CantanteFamoso a la lista. Usando iterator muestra los nombres de cada cantante y su disco con más ventas por pantalla. Se debe pedir al usuario un nombre y disco con más ventas de otro cantante famoso, y una vez introducidos los datos mostrar la lista actualizada usando iterator. Una vez mostrada la lista actualizada, se debe dar opción a elegir entre volver a introducir los datos de otro cantante o salir del programa (se podrán introducir tantos datos de cantantes como se desee. Para ello usa un bucle while que dé opción a elegir al usuario). Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Ejemplo de lo que podría ser la ejecución del programa:

La lista inicial contiene los siguientes datos:

Cantante: Madonna. Disco con más ventas: All I want is you.

Cantante: Jorge Negrete Disco con más ventas: Jalisco.

Por favor introduzca los datos de otro cantante.

Nombre: Michael Jackson

Disco con más ventas: Thriller

La lista actualizada contiene los siguientes datos:

Cantante: Madonna. Disco con más ventas: All I want is you.

Cantante: Jorge Negrete Disco con más ventas: Jalisco.

Cantante: Michael Jackson Disco con más ventas: Thriller.

¿Desea introducir los datos de otro cantante (s/n)?

s

Por favor introduzca los datos de otro cantante.

Nombre: Luis Miguel

Disco con más ventas: Mi jardín oculto

La lista actualizada contiene los siguientes datos:

Cantante: Madonna. Disco con más ventas: All I want is you.

Cantante: Jorge Negrete Disco con más ventas: Jalisco.

Cantante: Michael Jackson Disco con más ventas: Thriller.

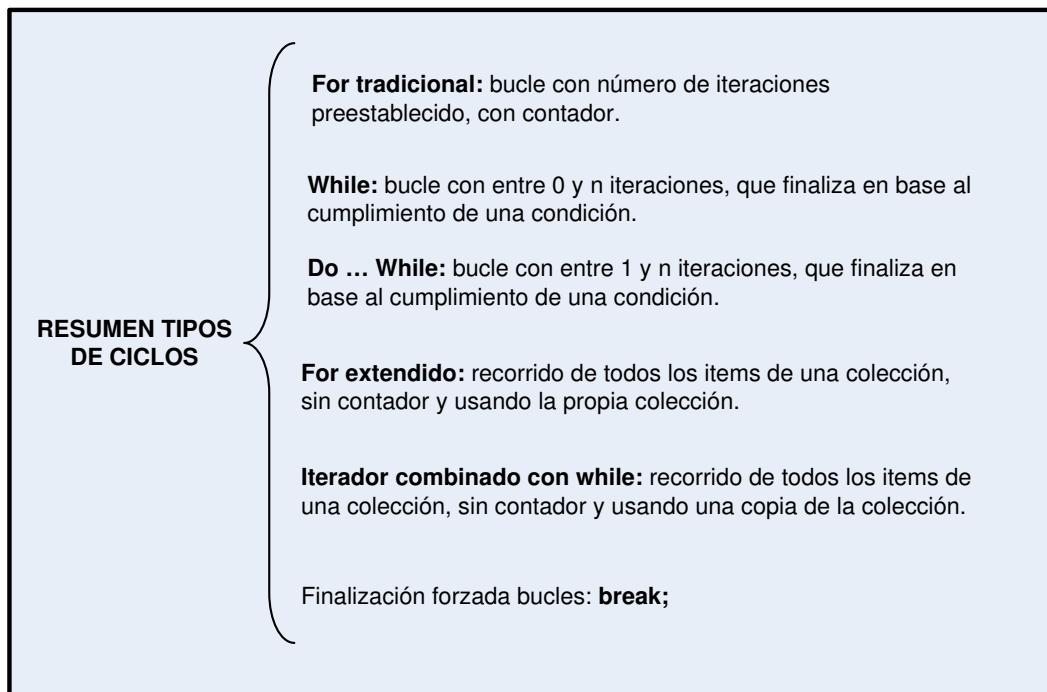
Cantante: Luis Miguel Disco con más ventas: Mi jardín oculto.

¿Desea introducir los datos de otro cantante (s/n)?

n

TIPOS DE BUCLES O CICLOS EN JAVA (RESUMEN)

Hasta ahora hemos visto distintas maneras de recorrer una colección o recorrer un bucle. El siguiente esquema es un resumen de ello.



Próxima entrega: CU00668B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

OBJETOS NULL Y JAVA.LANG.NULLPOINTEREXCEPTION EN JAVA

La palabra reservada “null” indica que una variable que referencia a un objeto se encuentra “sin objeto”, es decir, la variable ha sido declarada pero no apunta a ningún objeto. Esto puede deberse a que no se haya inicializado asignándole un objeto mediante la instrucción *new*, o a que hayamos borrado la referencia que contenía la variable.



Si a un objeto no inicializado (referencia *null*) se le trata de aplicar un método o se intenta hacerlo intervenir en un proceso que requiere un objeto inicializado, se obtiene una excepción tipo “NullPointerException”. Vamos a forzar que aparezca una excepción de este tipo. Para ello, crea dos clases y escribe el código que indicamos a continuación en cada una de ellas.

```
// Ejemplo aprenderaprogramar.com
import java.util.ArrayList;
public class ListaNumeros {
    private ArrayList<Integer> listaDeNumeros;

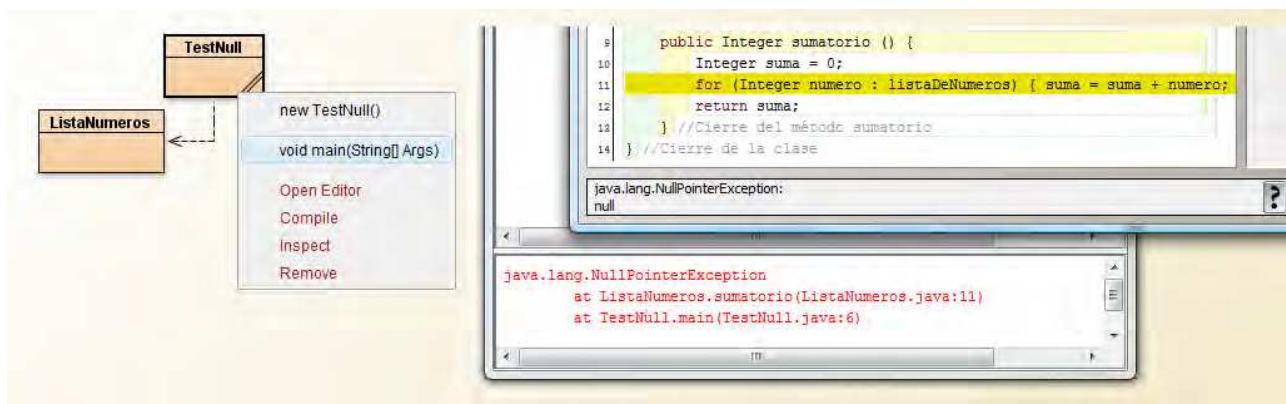
    public ListaNumeros () {} // El constructor está vacío

    public Integer sumatorio () {
        Integer suma = 0;
        for (Integer numero : listaDeNumeros) { suma = suma + numero; }
        return suma;
    } //Cierre del método sumatorio

    public void añadirItem (Integer item) {
        listaDeNumeros.add (item);
    } //Cierre del método
} //Cierre de la clase
```

```
// Ejemplo aprenderaprogramar.com
public class TestNull {
    public static void main (String [ ] Args) {
        ListaNumeros lista1 = new ListaNumeros();
        System.out.println ("El sumatorio actual es: " + lista1.sumatorio());
    } //Cierre del main
} //Cierre de la clase
```

Ejecuta ahora la clase que contiene el método main. El resultado será algo así:



El programa se empieza a ejecutar y en un momento determinado, se detiene (error en tiempo de ejecución). En la ventana del editor nos aparece `java.lang.NullPointerException` y en la ventana de consola nos aparecen referencias a las líneas de las clases en las que ha saltado el error. ¿A qué se debe este error? Java empieza a ejecutar el programa y reserva memoria para un objeto `ListaDeNumeros`. Como el constructor de esa clase está vacío, la variable tiene como referencia `null`. Al tratar de hacer un recorrido con un `for` extendido salta el error que indica que no se puede aplicar ese proceso sobre una variable que referencia a `null`. Modifica el constructor introduciendo este código: `public ListaNumeros () { listaDeNumeros = new ArrayList<Integer> (); }`

Comprueba que ahora el programa se ejecuta sin que salte ningún error. **Los errores de tipo `NullPointerException` ocurren con frecuencia** al tratar de añadir un ítem a una colección (lista, conjunto, etc.) sin haber inicializado explícitamente esa colección. No basta con que exista una declaración de la variable: hay que inicializarla.

Recordar: una variable (campo) que referencia a un objeto y no ha sido inicializada explícitamente contiene por defecto el valor `null`. En ocasiones podemos comprobar si una variable referencia a un objeto inicializado haciendo comprobaciones de este tipo:

```
if (ofertaMaxima == null) { instrucciones A } else { instrucciones B }
```

Este tipo de comprobaciones constituyen una medida de seguridad para evitar que se pueda intentar realizar un proceso no permitido como aplicar un método a una variable que contiene `null`.

AUTOBOXING Y UNBOXING. CONVERSIÓN AUTOMÁTICA DE TIPOS ENVOLTORIO A PRIMITIVOS Y VICEVERSA.

Comentamos en su momento que cada tipo primitivo tenía un tipo equivalente objeto. A los tipos objeto los denominábamos envoltorios. Dado que las colecciones como `ArrayList` contienen objetos y no tipos primitivos, puede parecer que esta circunstancia complica la programación si queremos por ejemplo trabajar con colecciones de números. Java solventa esta situación realizando una conversión

automática desde el tipo primitivo al envoltorio (autoboxing) o viceversa (unboxing). En el programa anterior son válidas expresiones como Integer suma = 0; ó lista1.add(7). El método add espera un objeto Integer. Sin embargo, le pasamos un tipo primitivo y lo acepta. Esto es debido al autoboxing que realiza el compilador Java en segundo plano evitando que salte un error.

El autoboxing y unboxing es una posibilidad destinada a facilitar el trabajo del programador, pero esto no elimina la necesidad de tener muy claro qué es un tipo primitivo y qué es un objeto y cuándo se requiere uno u otro. Comprobarás que no se puede, y no se debe, tratar de usar tipos primitivos y envoltorios como si se tratara de la misma cosa.

OBJETOS ANÓNIMOS

Consideremos el siguiente código:

```
//Ejemplo aprenderaprogramar.com
ListaNumeros lista1 = new ListaNumeros();
Integer num1 = new Integer (6);
lista1.add(num1);
lista1.add(7);
```

Cuando añadimos el objeto num1, conocemos el nombre del objeto que añadimos. Cuando añadimos mediante autoboxing un Integer que contiene el número 7, desconocemos el nombre de ese objeto. **Estos objetos que no tienen nombre específico se denominan “objetos anónimos”.** El uso de objetos anónimos es frecuente en toda situación en la que carece de interés disponer de nombres específicos para los objetos. La forma de crear objetos anónimos admite distintas sintaxis. Estos son algunos ejemplos:

```
//Ejemplos aprenderaprogramar.com
grupoDepositos.add (new Deposito (diametroNuevo, alturaNuevo, idNuevo) );
libreria.add (new Libro ("Elogio de la locura", "Erasmo de Rotterdam");
club.add (new Persona ("Juan Romero Sánchez", 32);
```

EJERCICIO

Responde a las siguientes preguntas. ¿Se puede acceder a un objeto anónimo contenido en un ArrayList? ¿Se puede inicializar un objeto de una clase que contiene tres atributos de tipo objeto declarando en el constructor que los atributos se inicializan mediante objetos anónimos? Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00669B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

ARRAYS, ARREGLOS O FORMACIONES EN JAVA

Algunas clases que hemos citado, como ArrayList o LinkedList, se denominan colecciones de tamaño flexible porque permiten modificar dinámicamente el número de ítems que contienen, es decir, ampliarlo o reducirlo. A veces nos referiremos a estas colecciones como arrays dinámicos porque son similares a los arrays dinámicos que se emplean en otros lenguajes de programación.



Los arrays dinámicos son muy potentes porque permiten crear colecciones de tamaño variable que podemos agrandar o empequeñecer en función de nuestras necesidades.

Sin embargo, cuando se conoce el número de elementos en una colección y éste va a ser invariable (por ejemplo, los 12 meses del año), será **más eficiente** utilizar una colección de tamaño fijo a la que denominamos array estático, arreglo estático, formación o vector. Si utilizamos el término array o arreglo, a secas, entenderemos que hacemos alusión a un array estático. El uso de arrays estáticos tiene ventajas e inconvenientes:

VENTAJAS	INCONVENIENTES
<ul style="list-style-type: none"> - Acceso y operación con elementos más eficiente. - Permiten almacenar tanto objetos como tipos primitivos directamente. 	<ul style="list-style-type: none"> - Rigidez. No se pueden redimensionar (aunque sí copiar a otros arrays estáticos de mayor tamaño).

La sintaxis a emplear con arrays la exponemos a continuación.

```

private TipoPrimitivoUObjeto [ ] nombreDelArray; //Declaración: reserva espacio de memoria

nombreDelArray = new TipoPrimitivoUObjeto [numero]; //Creación del array

private int [ ] miArrayDeNumeros = { 2, -3, 4, 7, -10 }; //Sintaxis ejemplo declarar y crear un array en una línea

// Ejemplo de uso con tipos primitivos aprenderaprogramar.com
private int [ ] cochesHorasDelDia;
cochesHorasDelDia = new int [24]; // Creamos un array de enteros con índices entre el 0 y el 23
cochesHorasDelDia [9] = 4521; //Ejemplo de asignación
cochesHorasDelDia [hora] = 4521; //Ejemplo de asignación
cochesHorasDelDia [23]++; //Ejemplo de asignación que equivale a cochesHorasDelDia[23] +=1;
private boolean[ ] superado;
superado = new boolean [1000];
superado [832] = false;
```

```
//Ejemplo de uso con tipos objeto aprenderaprogramar.com
private Persona [ ] Grupo3A;
Grupo3A = new Persona [50]; //Creamos un array de objetos Persona con índices entre el 0 y el 49
private String [ ] nombre;
nombre = new String [200];
nombre [37] = "Juan Antonio";

//Declarar el array y crear el objeto en una misma línea
Tipo [ ] nombreDelArray = new Tipo [número];

//Ejemplo de código que podríamos incluir en un método main aprenderaprogramar.com
int [ ] arrayEnteros = {2, 3, 1, 7, -1}; //El 2 es el elemento con índice 0 y el -1 el elemento con índice 4
String [ ] misNombres = new String [10];
misNombres [4] = "José Alberto Pérez";
System.out.println (misNombres[4]);
```

La numeración de los índices de los arrays va desde cero hasta (número de elementos – 1). Tener en cuenta que la variable que es el nombre del array, p.ej. misNombres, lo que contiene es un puntero o referencia al objeto que es en sí el array. **Un array en Java puede considerarse un “objeto especial”**. Se crea con la sentencia *new* como el resto de objetos, pero sin embargo no hay una clase específica en Java que defina el tipo de los arrays. Dada una declaración del tipo *int [] cochesPorHora = new int [24];*, hay ciertos errores habituales frente a los que hay que estar atentos:

- Pensar que los índices van de 0 a 24. Falso: van de 0 a 23.
- Pensar que el número de elementos total es 23. Falso: son 24.
- Usar *cochesPorHora [24]*. El índice 24 no existe y el uso de esa expresión daría lugar a un error en tiempo de ejecución del tipo “*ArrayIndexOutOfBoundsException*”.

Sobre un objeto que es un elemento de un array se pueden invocar métodos. Por ejemplo *System.out.println (persona[17].getNombre());*. Por otro lado, un elemento de un array puede almacenar un objeto anónimo. Por ejemplo *persona [47] = new Persona ("Juan", "Pérez Hernández", 39, 1.75);*, donde los elementos entre paréntesis son los parámetros requeridos por el constructor de la clase Persona.

Para recorrer arrays, dado que se conoce el número de iteraciones, es habitual usar *for* tradicionales.

CAMPO LENGTH PARA SABER EL NÚMERO DE ELEMENTOS DE UN ARRAY

La sintaxis *nombreDelArray.length* nos devuelve un entero (int) con el número de elementos que forman el array. Fíjate que después de length no aparecen paréntesis, lo que indica que no estamos invocando un método, sino accediendo a un atributo del array. El acceso a este atributo es posible porque el API de Java mantiene este atributo como público: **si fuera privado no podríamos acceder a él**. Ejemplos de uso podrían ser los siguientes:

```
System.out.println ("El número de elementos en el array misNombres es de " + misNombres.length );
for (int i = 0; i < misNombres.length; i++) { System.out.println ("Nombre " + i + ": " + misNombres[i]); }
```

En este ejemplo usamos `i <`, es decir, menor estricto, porque `length` nos devuelve siempre un valor `n+1` para unos índices comprendidos entre `0` y `n`.

USO DE CICLOS FOR EACH CON ARRAYS

Es posible usar ciclos `for – each` con arrays. Por ejemplo:

```
//Ejemplo aprenderaprogramar.com
for (int tmpItem : cochesPorHora) { System.out.println ("Número: " + tmpItem); }

String [ ] misNombres = new String [10];
For (String tmpObjeto : misNombres) { System.out.println (tmpObjeto); }
//Nota: en el caso de que un ítem objeto no tenga contenido por pantalla saldrá null
```

El `for each` tiene ventajas como el poder recorrer una colección de la que se desconoce su tamaño, pero también inconvenientes como carecer de una variable contadora en el ciclo. Si quisieramos contar tendríamos que introducir una variable contadora, cosa que con el `for normal` ya tenemos con la propia definición del bucle. Dado que en un array podemos conocer con facilidad el número de elementos de que consta y disponer de forma automática de la variable contadora, será más habitual el uso de `for tradicionales` con arrays que el uso del `for extendido`.

EJERCICIO

Crea una clase con el método `main` donde declares una variable de tipo array de `Strings` que contenga los doce meses del año, en minúsculas y declarados en una sola línea. A continuación declara una variable `mesSecreto` de tipo `String`, y hazla igual a un elemento del array (por ejemplo `mesSecreto = mes[3]`). El programa debe pedir al usuario que adivine el mes secreto y si acierta mostrar un mensaje y si no pedir que vuelva a intentar adivinar el mes secreto. Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Un ejemplo de ejecución del programa podría ser este:

Adivine el mes secreto. Introduzca el nombre del mes en minúsculas: **febrero**

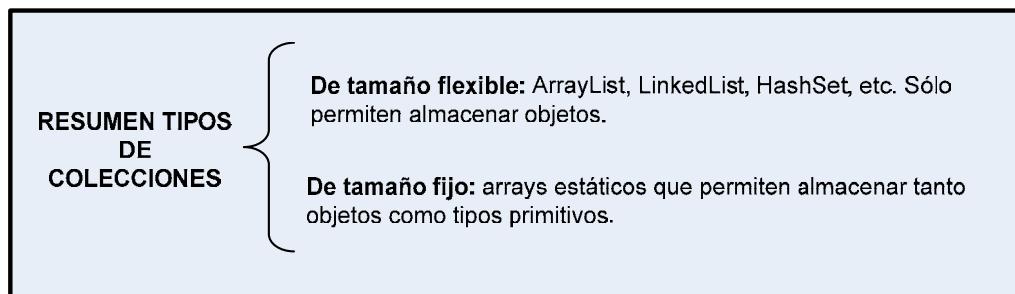
No ha acertado. Intente adivinarlo introduciendo otro mes: **agosto**

No ha acertado. Intente adivinarlo introduciendo otro mes: **octubre**

¡Ha acertado!

RESUMEN DE TIPOS DE COLECCIONES JAVA

El siguiente esquema resume los tipos de colecciones disponibles en Java.



No hay que usar con preferencia uno u otro tipo. Para cada caso, conviene estudiar las circunstancias y elegir el más adecuado.

Próxima entrega: CU00670B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONVERSIÓN DE TIPOS DE DATOS EN JAVA

En Java es posible transformar el tipo de una variable u objeto en otro diferente al original con el que fue declarado. Este proceso se denomina "**conversión**", "**moldeado**" o "**tipado**" y es algo que debemos manejar con cuidado pues un mal uso de la conversión de tipos es frecuente que dé lugar a errores.



Una forma de realizar conversiones consiste en colocar el tipo destino entre paréntesis, a la izquierda del valor que queremos convertir de la forma siguiente: `Tipo VariableNueva = (NuevoTipo) VariableAntigua;`

Por ejemplo: `int miNúmero = (int) ObjetoInteger; char c = (char)System.in.read();`

En el primer ejemplo, extraemos como tipo primitivo int el valor entero contenido en un campo del objeto Integer. En el segundo caso, la función read devuelve un valor int, que se convierte en un char debido a la conversión (char), y el valor resultante se almacena en la variable de tipo carácter c.

El tamaño de los tipos que queremos convertir es muy importante. No todos los tipos se convertirán de forma segura. Por ejemplo, al convertir un long en un int, el compilador corta los 32 bits superiores del long (de 64 bits), de forma que encajen en los 32 bits del int, con lo que si contienen información útil, ésta se perderá. Este tipo de conversiones que suponen pérdida de información se denominan "**conversiones no seguras**" y en general se tratan de evitar, aunque de forma controlada pueden usarse puntualmente.

De forma general trataremos de atenernos a la norma de que "**en las conversiones debe evitarse la pérdida de información**". En la siguiente tabla vemos conversiones que son seguras por no suponer pérdida de información.

TIPO ORIGEN	TIPO DESTINO
<code>byte</code>	<code>double, float, long, int, char, short</code>
<code>short</code>	<code>double, float, long, int</code>
<code>char</code>	<code>double, float, long, int</code>
<code>int</code>	<code>double, float, long</code>
<code>long</code>	<code>double, float</code>
<code>float</code>	<code>Double</code>

No todos los tipos se pueden convertir de esta manera. Como alternativa, existen otras formas para realizar conversiones.

MÉTODO VALUEOF PARA CONVERSIÓN DE TIPOS

El método `valueOf` es un método sobrecargado aplicable a numerosas clases de Java y que permite realizar conversiones de tipos. Veamos algunos ejemplos de uso.

EXPRESIÓN	INTERPRETACIÓN aprenderaprogramar.com
<code>miInteger = miInteger.valueOf (i)</code>	Con i entero primitivo que se transforma en Integer
<code>miInteger = miInteger.valueOf (miString)</code>	El valor del String se transforma en Integer
<code>miString = miString.valueOf (miBooleano)</code>	El booleano se transforma en String “true” o “false”
<code>miString = miString.valueOf (miChar)</code>	El carácter (char) se transforma en String
<code>miString = miString.valueOf (miDouble)</code>	El double se transforma en String. Igualmente aplicable a float, int, long.

No todas las conversiones son posibles. Muchas veces por despiste los programadores escriben instrucciones de conversión incoherentes como `miInteger = (int) miString;`. El resultado en este caso es un error de tipo “Inconvertible types”. Un uso típico de `valueOf` es para convertir tipos primitivos en objetos.

EJERCICIO

El API de Java proporciona herramientas para pedir datos al usuario a través de ventanas. Un ejemplo de ello es el uso de la clase `JOptionPane` perteneciente al paquete `javax.swing.JOptionPane` del API Java. El método `showInputDialog` permite pedir un dato al usuario y almacena su respuesta en un objeto de tipo `String`. Queremos crear un programa que pida al usuario un número y muestre por pantalla el doble de ese número. Para ello hemos creado este código:

```
import javax.swing.JOptionPane;
public class ejemplo1 {
    public static void main (String[] Args) {
        String entradaUsuario = JOptionPane.showInputDialog ( "Introduzca un número:" );
        System.out.println ("El doble del número introducido es: " + 2*entradaUsuario);
    }
}
```

El problema que tenemos es que nos salta un mensaje de error “operator * cannot be applied to int, java.lang.String”. ¿Qué interpretación haces de este mensaje de error? ¿Cómo se puede corregir el código para que a través de una conversión de tipos se ejecute el programa? Puedes comprobar si tu código y respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00671B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

GET Y REMOVE DE ARRAYLIST. EJEMPLO CONVERSIÓN DE TIPOS.

Cuando utilizamos llamadas a métodos hemos de prestar especial atención al tipo requerido por los parámetros y al tipo devuelto (si se trata de un método tipo función). Si no lo hacemos, nuestros programas pueden dar lugar a errores o resultados incoherentes. Escribe y compila el siguiente código.



```
//Test conversión de tipos aprenderaprogramar.com
import java.util.ArrayList;

public class TestConversion {
    public static void main (String [ ] Args) {

        ArrayList<Integer> miListadoAL;
        miListadoAL = new ArrayList<Integer>();
        ArrayList<Integer> tmpAL = new ArrayList<Integer>();
        miListadoAL.add(44); miListadoAL.add(7); miListadoAL.add(76); miListadoAL.add(29); miListadoAL.add(17);

        //Recorremos el ArrayList con un for normal quedándonos con los elementos que van a ser operados a posteriori
        for (int i=1; i <= miListadoAL.size(); i++) {
            if (miListadoAL.get(i-1) < 10) { System.out.println ("Detectado un objeto (" + i + ") :" +miListadoAL.get(i-1) + ", integer con valor menor de 10, será eliminado");
                tmpAL.add(i-1);
                System.out.println ("Elemento en el array list (" + i + ") : " + miListadoAL.get(i-1));
            }
        }

        for (int i=0; i < tmpAL.size(); i++) {
            System.out.println ("Eliminamos ítem "+ (tmpAL.get(i)+1) +": " + miListadoAL.get(tmpAL.get(i) ) );
            miListadoAL.remove(tmpAL.get(i));
        }

        System.out.println("Valor devuelto por size() en miListadoAL después de borrado: " + miListadoAL.size());
        int tmpInt = 0;
        for (Integer tmpObjeto: miListadoAL) {
            System.out.println ("Elemento en el array list ("+(tmpInt+1)+") : " + tmpObjeto);
            tmpInt++;
        }

    } //Cierre del main
} //Cierre de la clase
```

Analiza y trata de comprender el código pues aplicamos conceptos que hemos ido estudiando previamente. Si no comprendes algo repasa los contenidos anteriores antes de seguir leyendo. El resultado será algo similar a esto:

```

Elemento en el array list (1) : 44
Detectado un objeto (2) :7, integer con valor menor de 10, será eliminado
Elemento en el array list (2) : 7
Elemento en el array list (3) : 76
Elemento en el array list (4) : 29
Elemento en el array list (5) : 17
Eliminamos ítem 2: 7
Valor devuelto por size() en miListadoAL después de borrado: 5
Elemento en el array list (1) : 44 Elemento en el array list (2) : 7
Elemento en el array list (3) : 76 Elemento en el array list (4) : 29
Elemento en el array list (5) : 17

```

Obviamente nuestra intención va por un lado y los resultados obtenidos por otro. Algo no está funcionando, y sin embargo no nos salta ningún error. ¿Qué está ocurriendo? Tenemos una disfunción entre el tipo requerido por un método y el tipo que le pasamos. Si consultas la documentación de los métodos `get` y `remove` de la clase `ArrayList`, comprobarás que el método `get` devuelve un objeto (`Integer`) mientras que el método `remove` requiere un tipo primitivo (`int`). Nosotros le estamos pasando al método `remove` lo que nos devuelve el método `get`, es decir, un `Integer`. Para que el programa responda adecuadamente hemos de pasar a los métodos los tipos adecuados. Por tanto, hemos de pasar al método `remove` un tipo `int` y para ello hemos de indicar específicamente la conversión del objeto `Integer` a un primitivo `int`. Corrige el código introduciendo la conversión de tipos:

```

//Ejemplo aprenderaprogramar.com
for (int i=0; i < tmpAL.size(); i++) {
    System.out.println ("Eliminamos ítem "+ (tmpAL.get(i)+1) +": " + miListadoAL.get(tmpAL.get(i) ) );
    miListadoAL.remove( int tmpAL.get(i) ); //Introducida conversión de tipos
}

```

El resultado ahora obtenido sí debe ser el esperado:

```

Elemento en el array list (1) : 44
Detectado un objeto (2) :7, integer con valor menor de 10, será eliminado
Elemento en el array list (2) : 7
Elemento en el array list (3) : 76
Elemento en el array list (4) : 29
Elemento en el array list (5) : 17
Eliminamos ítem 2: 7
Valor devuelto por size() en miListadoAL después de borrado: 4
Elemento en el array list (1) : 44 Elemento en el array list (2) : 76
Elemento en el array list (3) : 29 Elemento en el array list (4) : 17

```

Este ejemplo ilustra la importancia de realizar un correcto manejo de tipos en la programación Java.

Próxima entrega: CU00672B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

GENERAR NÚMEROS ALEATORIOS EN JAVA

Hay varias maneras de generar números aleatorios en Java. Entre ellas, **métodos previstos por las clases Math y Random**. La terminología puede resultarnos inicialmente confusa, pues la clase Math dispone de un método denominado random(), mientras que Random es a su vez el nombre de otra clase.



La generación de números aleatorios por ordenador no es tan sencilla como pueda parecer debido a que la operación de los ordenadores es determinística: se apoya en el hecho de que todo cálculo es predecible y repetible. Por ello conviene tener presente que sería más apropiado usar el término “pseudealeatorio” para referirnos a los números aparentemente aleatorios que podemos obtener de un ordenador.

Vamos a ver un extracto de la documentación de la clase Random (ten en cuenta que pueden existir pequeñas diferencias según la versión de Java que estés utilizando). Para utilizarla habremos de indicar en cabecera `import java.util.Random;` o de forma más general `import java.util.*;`. Para generar números (pseude) aleatorios vamos a tener que crear un objeto de tipo Random y luego invocar un método sobre ese objeto que nos devuelva el (pseude) aleatorio. Veamos el extracto de documentación:

`java.util`

Class Random

```
java.lang.Object
└─ java.util.Random
```

Un objeto de esta clase se usa para generar una secuencia (stream) de números pseudealeatorios a partir de un valor inicial o “semilla”. Si dos instancias de la clase Random son creadas con la misma “semilla”, y se ejecutan las mismas instrucciones para ambas instancias, se generarán y devolverán idénticas secuencias de números.

Constructor Summary (se omiten algunos constructores)

<code>Random()</code>	Crea un generador de números aleatorios. Usando este constructor, en cada invocación Java se encarga de que la “semilla” para generación de aleatorios sea siempre distinta.
-----------------------	--

Method Summary (sólo algunos métodos, otros se omiten)

boolean	nextBoolean() Devuelve un valor true o false de forma pseudoaleatoria. La probabilidad de obtener un valor u otro es de aproximadamente la mitad.
double	nextDouble() Devuelve un pseudoaleatorio de tipo double, uniformemente distribuido entre 0.0 y 1.0.
float	nextFloat() Devuelve un pseudoaleatorio de tipo float, uniformemente distribuido entre 0.0 y 1.0.
int	nextInt() Devuelve un pseudoaleatorio de tipo int con valores entre 0 y 2^{32} producidos con aproximadamente la misma probabilidad.
int	nextInt(int n) Devuelve un pseudoaleatorio de tipo int comprendido entre cero (incluido) y el valor especificado (excluido).
long	nextLong() Devuelve un pseudoaleatorio de tipo long.
Otros	Existen más métodos (consultar la documentación del API de Java para ampliar información)

Vamos a escribir un programa que utiliza la clase Random. Además también utiliza el print en lugar del println y el carácter de escape \n para forzar cambio de línea y retorno de carro. Utilizaremos la clase EntradaDeTeclado que ya hemos escrito y usado en programas anteriores, aunque con una pequeña variación en el constructor. Escribe el código de las siguientes clases.

```
import java.util.Scanner; //Importación de la clase Scanner desde la biblioteca Java
public class EntradaDeTeclado { // Definimos la clase EntradaDeTeclado aprenderaprogramar.com
    private String entradaTeclado; //Variable de instancia (campo) del método

    //Constructor
    public EntradaDeTeclado () {
        entradaTeclado = ""; } //Cierre del constructor

    public void pedirEntrada () { //Método de la clase
        Scanner entradaEscaner = new Scanner (System.in);
        entradaTeclado = entradaEscaner.nextLine ();
    } //Cierre del método pedirEntrada

    public String getEntrada () {
        return entradaTeclado;
    } //Cierre del método getEntrada

} //Cierre de la clase
```

```

import java.util.ArrayList; import java.util.Random; //Ejemplo aprenderaprogramar.com

//Esta clase define objetos que contienen tantos enteros aleatorios entre 0 y 1000 como se definan
public class SerieDeAleatorios {
    //Campos de la clase
    private ArrayList<Integer> serieAleatoria;

    //Constructor
    public SerieDeAleatorios (int numeroItems) {
        serieAleatoria = new ArrayList<Integer> ();
        //Inicializamos el ArrayList lleno de ceros
        for (int i=0; i<numeroItems; i++) { serieAleatoria.add(0); }
        System.out.println ("Serie inicializada. El número de elementos en la serie es: " + getNumeroItems() );
    } //Cierre del constructor

    public int getNumeroItems() { return serieAleatoria.size(); } //Cierre del método

    public void generarSerieDeAleatorios () {
        Random numAleatorio;
        numAleatorio = new Random ();
        for (int i=0; i < serieAleatoria.size(); i++) {
            serieAleatoria.set(i, numAleatorio.nextInt(1000) );
        }
        System.out.print ("Serie generada! ");
    } //Cierre del método

    public void mostrarSerie() {
        System.out.print ("Procedemos a mostrar la serie: ");
        for (Integer tmpObjeto : serieAleatoria) { //Uso de for each
            System.out.print (" " + tmpObjeto.toString() ); } //Cierre del for extendido
        } //Cierre del método
    } //Cierre de la clase
}

```

```

// Programa Test de obtención de números pseudoaleatorios aprenderaprogramar.com
public class TestPseudoAleatorios {
    public static void main (String [] Args) {
        Integer tmpInteger = 0;
        EntradaDeTeclado entradaMain = new EntradaDeTeclado();
        char tecla = 'S';
        while (tecla == 'S') {
            System.out.print ("Por favor introduzca el número de elementos en la serie de números aleatorios: ");
            entradaMain.pedirEntrada();
            tmpInteger = tmpInteger.valueOf (entradaMain.getEntrada() );
            SerieDeAleatorios serieDePrueba = new SerieDeAleatorios ( (int) tmpInteger);
            serieDePrueba.generarSerieDeAleatorios();
            serieDePrueba.mostrarSerie();
            tecla = ' ';
            System.out.println ("\n¿Generar otra serie (S/N):");
            while (tecla != 'n' && tecla != 'N' && tecla != 's' && tecla != 'S') {
                entradaMain.pedirEntrada();
                //Ojo tenemos que comparar usando equals porque los string son objetos!!!!
                if (entradaMain.getEntrada().equals("n") || entradaMain.getEntrada().equals("N") ) { tecla = 'N'; }
                else if (entradaMain.getEntrada().equals("s") || entradaMain.getEntrada().equals("S") ) { tecla = 'S'; }
            } //Cierre del while interior
        } //Cierre del while exterior
        System.out.println ("Gracias por utilizar el programa");
    } } //Cierre del main y de la clase
}

```

El resultado del programa será similar a este:

```
Por favor introduzca el número de elementos en la serie de números aleatorios: 3
Serie inicializada. El número de elementos en la serie es: 3
Serie generada! Procedemos a mostrar la serie: 745 777 24
¿Generar otra serie (S/N):
S
Por favor introduzca el número de elementos en la serie de números aleatorios: 7
Serie inicializada. El número de elementos en la serie es: 7
Serie generada! Procedemos a mostrar la serie: 314 520 402 944 71 248 839
¿Generar otra serie (S/N):
N
Gracias por utilizar el programa
```

EJERCICIO

Crea un programa Java que permita “jugar a adivinar un número” como se expone a continuación. El programa debe iniciarse indicando “Se ha generado un número aleatorio entero entre 1 y 100, intente adivinarlo”. El usuario introducirá un número y si el número aleatorio generado por el ordenador es menor deberá indicarse “No has acertado: el número es menor. Prueba otra vez”. Si el usuario introduce un número menor que el número aleatorio deberá indicarse “No has acertado: el número es mayor”. El programa terminará cuando el usuario introduzca el número aleatorio que había escogido el ordenador. Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00673B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PALABRAS CLAVE STATIC Y FINAL. CONSTANTES EN JAVA.

En los programas que generemos usualmente intervendrán constantes: valores matemáticos como el número Pi, o valores propios de programa que nunca cambian. Si nunca cambian, lo adecuado será declararlos como constantes en lugar de cómo variables. Supongamos que queremos usar una constante como el número Pi y que usamos esta declaración:



```
// Ejemplo aprenderaprogramar.com
public class Calculadora {
    private double PI = 3.1416;
    public void mostrarConstantePi () { System.out.println (PI); }
    ... constructor, métodos, ... código de la clase ... }
```

Si creas un objeto de tipo Calculadora, comprobarás que puedes invocar el método mostrarConstantePi para que te muestre el valor por pantalla. No obstante, una declaración de este tipo presenta varios problemas. En primer lugar, lo declarado no funciona realmente como constante, sino como variable con un valor inicial. Prueba a establecer *this.PI* = 22; dentro del método y verás que es posible, porque lo declarado es una variable, no una constante. En segundo lugar, cada vez que creamos un objeto de tipo calculadora estamos usando un espacio de memoria para almacenar el valor 3.1416. Así, si tuviéramos diez objetos calculadora, tendríamos diez espacios de memoria ocupados con la misma información, lo cual resulta ineficiente. Para resolver estos problemas, podemos declarar constantes en Java usando esta sintaxis:

Carácter Público/Privado static final TipoDeLaConstante = valorDeLaConstante;

En esta declaración intervienen dos palabras clave cuyo significado es importante:

- static:** los atributos miembros de una clase pueden ser atributos de clase o atributos de instancia; se dice que son atributos de clase si se usa la palabra clave *static*: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria). A veces a las variables de clase se les llama variables estáticas. Si no se usa *static*, el sistema crea un lugar nuevo para esa variable con cada instancia (la variable es diferente para cada objeto). En el caso de una constante no tiene sentido crear un nuevo lugar de memoria por cada objeto de una clase que se cree. Por ello es adecuado el uso de la palabra clave *static*. Cuando usamos “*static final*” se dice que creamos una constante de clase, un atributo común a todos los objetos de esa clase.

- b) **final:** en este contexto indica que una variable es de tipo constante: no admitirá cambios después de su declaración y asignación de valor. *final* determina que un atributo no puede ser sobreescrito o redefinido. O sea: no funcionará como una variable “tradicional”, sino como una constante. Toda constante declarada con *final* ha de ser inicializada en el mismo momento de declararla. *final* también se usa como palabra clave en otro contexto: una clase final (*final*) es aquella que no puede tener clases que la hereden. Lo veremos más adelante cuando hablemos sobre herencia.

Cuando se declaran constantes es muy frecuente que los programadores usen letras mayúsculas (como práctica habitual que permite una mayor claridad en el código), aunque no es obligatorio.

Una declaración en cabecera de una clase como *private final double PI = 3.1416;* podríamos interpretarla como una constante de objeto. Cada objeto tendrá su espacio de memoria con un contenido invariable. Una declaración en cabecera de una clase como *private static final double Pi = 3.1416;* la interpretamos como una constante de clase. Existe un único espacio de memoria, compartido por todos los objetos de la clase, que contiene un valor invariable. Veamos ejemplos de uso:

```
// Ejemplo aprenderaprogramar.com
// Sintaxis: CaracterPublico/Privado static final TipoDeLaConstante = valorDeLaConstante;
private static final float PI = 3.1416f; //Recordar f indica que se trata de un float
private static double PI = 3.1416;
public static final String passwd = "jkl342lagg";
public static final int PRECIO_DEFAULT = 100;
```

Cuando usamos la palabra clave *static* la declaración de la constante ha de realizarse **obligatoriamente en cabecera de la clase**, junto a los campos (debajo de la firma de clase). Es decir, un atributo de clase hemos de declararlo en cabecera de clase. Si tratamos de incorporarlo en un método obtendremos un error. Por tanto dentro del método main (que es un método de clase al llevar incorporado static en su declaración) no podemos declarar constantes de clase. Por otro lado, *final* sí puede ser usado dentro de métodos y también dentro de un método main. Por ejemplo una declaración como *final String psswd = "mt34rsm8"* es válida dentro de un método. En resumen: en cabecera de clase usaremos *static final* para definir aquellas variables comunes a todos los objetos de una clase.

Modifica el código de la clase Calculadora que vimos anteriormente para declarar PI como una constante de clase. Luego, intenta modificar el valor de PI usando una invocación como *this.PI =22;*. Comprobarás que se produce un error al compilar ya que los valores de las constantes no son modificables.

EJERCICIO

Define una clase Java denominada Circulo que tenga como atributo de clase (estático) y constante numeroPi, siendo esta constante de tipo double y valor 3.1416. Además la clase tendrá el atributo radio (tipo double) que representa el radio del círculo, y los métodos para obtener y establecer los atributos. También debe disponer de un método para calcular el área del círculo (método tipo función areaCirculo que devuelve el área) y la longitud del círculo (método tipo función que devuelve la longitud). Busca información sobre las fórmulas necesarias para crear estos métodos en internet si no las recuerdas. En una clase con el método main, declara el código que cree un objeto círculo, le pida al usuario el radio y le devuelva el área y la longitud del círculo.

¿Es posible crear un método en la clase Circulo para establecer el valor de numeroPi? ¿Por qué?

Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00674B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

PROYECTOS JAVA EN PAQUETES (PACKAGES)

Los proyectos (en general podríamos establecer equivalencia de proyecto con programa, pero no siempre es así; proyecto es un conjunto de código que se mantiene agrupado) en Java se suelen organizar en paquetes (packages). El concepto de paquete viene siendo similar al de carpeta en Windows: un contenedor donde mantenemos cosas relacionadas entre sí.



De hecho, en entornos como Eclipse o BlueJ esto es exactamente así: al crear packages, veremos que creamos una carpeta. No obstante, la organización en packages tiene muchas más implicaciones como veremos a continuación.

La organización del proyecto será por tanto similar a la organización de archivos: en un paquete podremos tener por ejemplo clases de tipo A, en otro clases de tipo B y así sucesivamente. A su vez, un paquete puede contener subpaquetes: por ejemplo el paquete A puede contener a los subpaquetes A.1, A.2 y A.3. **Un package es una agrupación de clases afines.** Recuerda el concepto de librería existente en otros lenguajes o sistemas.

Una clase puede definirse como perteneciente a un package y puede usar otras clases definidas en ese o en otros packages. Los packages delimitan el espacio de nombres (space name). El nombre de una clase debe ser único dentro del package donde se define. Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser usadas en el mismo programa. Una clase se declara perteneciente a un package con la cláusula package incluida como una línea de código, cuya sintaxis es: *package nombrePackage;*. La cláusula *package* debe ser la primera sentencia del código fuente. Por ejemplo, una clase puede comenzar así:

```
package miPackage;  
...  
public class miClase {  
...}
```

Este código implica que la clase miClase pertenece al package miPackage. La cláusula *package* es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto (sin nombre). La agrupación de clases en packages es conveniente desde el punto de vista organizativo, para mantener bajo una ubicación común clases relacionadas que cooperan desde algún punto de vista. También **resulta importante por la implicación que los packages tienen en la visibilidad y acceso del código** entre distintas partes de un programa. Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
package Directivos;
...
public class JefeDeAdministracion { //Ejemplo aprenderaprogramar.com
    private Persona jefeAdmin;
    ...
}
```

En esta declaración definimos la clase `JefeDeAdministracion` perteneciente al package `Directivos`. Esta clase usa la clase `Persona`. El compilador asume que `Persona` pertenece también al package `Directivos`, y tal como está hecha la definición, para que la clase `Persona` sea accesible (conocida) por el compilador, es necesario que esté definida en el mismo package. Si esto no fuera así, es necesario hacer accesible el espacio de nombres donde está definida la clase `Persona` a nuestra nueva clase. Esto se hace con la cláusula `import`. Supongamos que la clase `Persona` estuviera definida de esta forma:

```
package TiposBasicos;
public class Persona {
    private String nombre;
}
```

Entonces, para usar la clase `Persona` en nuestra clase `JefeDeAdministracion` deberíamos poner:

```
package Directivos;
import TiposBasicos.*;
class JefeDeAdministracion {
    private Persona jefeAdmin;
    ...
}
```

Con la cláusula `import TiposBasicos.*;` se hacen accesibles todas las clases declaradas en el package `TiposBasicos`. Si sólo se quisiera tener accesible la clase `Persona` se podría declarar: `import TiposBasicos.Persona;`. También es posible hacer accesibles los nombres de un package sin usar la cláusula `import` calificando completamente los nombres de aquellas clases pertenecientes a otros packages. Por ejemplo:

```
package Directivos;
class JefeDeAdministracion { //Ejemplo aprenderaprogramar.com
    private TiposBasicos.Persona jefeAdmin;
    ...
}
```

Sin embargo si no se usa `import` es necesario especificar el nombre del package cada vez que se usa el tipo `Persona`, lo cual puede resultar un tanto engorroso. La cláusula `import` en esta acepción simplemente indica al compilador dónde debe buscar clases adicionales cuando no pueda encontrarlas en el package actual y delimita los espacios de nombres y modificadores de acceso. Sin embargo, no tiene la implicación de 'importar' o copiar código fuente u objeto alguno. **En una clase puede haber tantas sentencias import como sean necesarias.** Las cláusulas `import` se colocan después de la cláusula `package` (si es que existe) y antes de las definiciones de las clases.

FORMAS DE NOMBRAR PACKAGES. JERARQUIZACIÓN Y VISIBILIDAD DE CLASES. BLUEJ.

Los packages se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las llamadas a los métodos. Por ejemplo se puede tener un package de nombre tiposBasicos.directivos. Cuando se utiliza esta estructura se habla de packages y subpackages. En el ejemplo tiposBasicos es el Package base, directivos es un subpackage y a su vez podrían existir otros subpackages de directivos. De esta forma se pueden tener los packages ordenados según una jerarquía equivalente a un sistema de archivos jerárquico. El API de Java está estructurado de esta forma, con un primer calificador (java o javax) que indica la base, un segundo calificador (awt, util, swing, etc.) que indica el grupo funcional de clases y opcionalmente subpackages en un tercer nivel, dependiendo de la amplitud del grupo. Por ejemplo en `java.util.ArrayList<E>`, `java` sería el package básico, `util` un subpackage y `ArrayList` una clase. Cuando se crean packages de usuario no es recomendable usar nombres de packages que empiecen por `java` o `javax` para evitar confusiones con los propios del API de Java.

Además del significado lógico descrito hasta ahora, **los packages también tienen un significado físico** al servir para almacenar los ficheros con extensión .class en el sistema de archivos del ordenador. Supongamos que definimos una clase de nombre miClase que pertenece a un package de nombre `gestorDePersonal.tiposBasicos.directivos`. Cuando la JVM vaya a cargar en memoria miClase buscará el módulo ejecutable (de nombre `miClase.class`) en un directorio en la ruta de acceso `gestorDePersonal/tiposBasicos/directivos` (esta ruta de acceso estará definida a partir del directorio raíz para la JVM). Esta ruta deberá existir y estar accesible a la JVM para que encuentre las clases.

Si una clase no pertenece a ningún package (no existe cláusula `package`) se asume que pertenece a un package por defecto sin nombre, y la JVM buscará el archivo .class en el directorio que use como raíz.

Para que una clase pueda ser usada fuera del package donde se definió debe ser declarada con el modificador de acceso `public`. Por ejemplo:

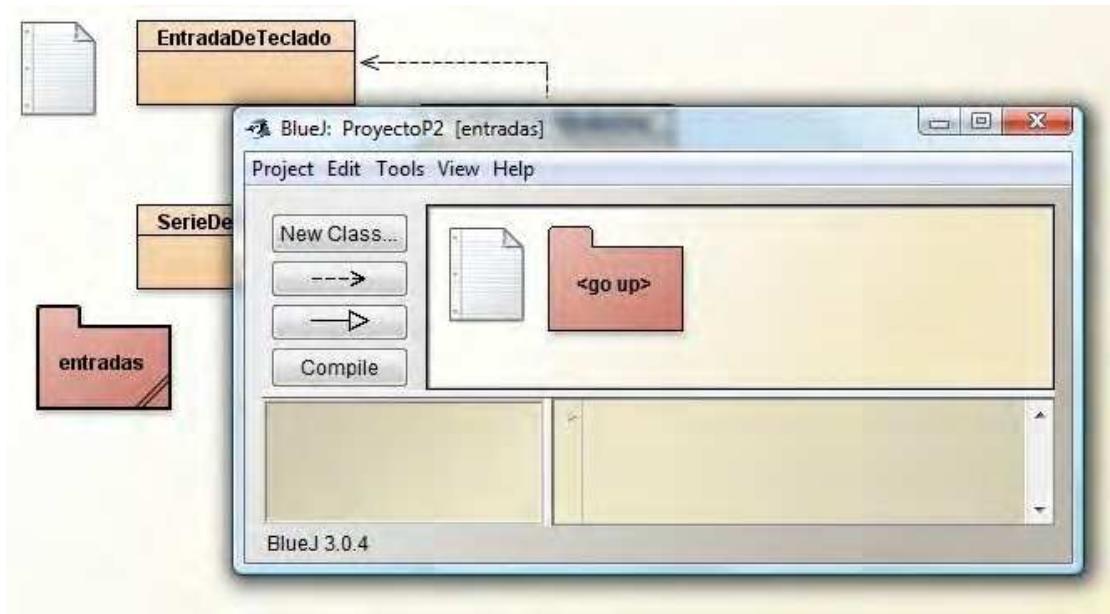
```
package Directivos;
public class JefeDeAdministracion {
    private Persona jefeAdmin;
    ...
}
```

Si una clase no se declara `public` sólo puede ser usada por clases que pertenezcan al mismo package.

Consideremos un proyecto en BlueJ. Hasta ahora lo hemos visto como algo así:



Aquí veíamos las clases y disponíamos del botón New Class... para ir añadiendo clases. ¿En qué package están estas clases? En el package por defecto (sin nombre), asociado al directorio raíz donde hayamos guardado el proyecto. Supongamos ahora que nuestra organización fuera que la clase EntradaPorTeclado perteneciera a un package denominado entradas.usuarios, es decir, que la clase perteneciera al subpaquete "usuarios" del paquete "entradas". Para crear los paquetes pulsamos en el menú Edit -> New package, y escribimos el nombre del package base, esto es, *entradas*. Nos aparece un icono "entradas" y haciendo doble click sobre él se nos abre la ventana del package creado:



Donde el icono tipo carpeta "go up" nos sirve para subir de nivel en la jerarquía de packages. Situados en la ventana del package *entradas* repetiremos la operación creando el package *usuarios*. Ahora desde el package *entradas* podemos hacer dos cosas: subir de nivel hacia la raíz, o profundizar nivel hacia *usuarios*. Si comprobamos nuestro sistema de archivos, veremos cómo se han ido creando carpetas anidadas que de momento sólo contienen un archivo package.BlueJ. Además, si entramos en el package *usuarios* y vemos lo que nos indica la ventana de BlueJ, aparece [*entradas.usuarios*], es decir, por defecto nos está estableciendo la ruta de packages con notación de puntos (muy al estilo de Java). En el nivel raíz, borremos ahora el package *entradas*. Nos aparecerá una advertencia indicándonos que si borramos el package eliminaremos de forma permanente el directorio asociado al package incluyendo todos los contenidos de ese directorio (esto tiene su peligro pues podríamos borrar carpetas accidentalmente). Aceptemos y vayamos al sistema de archivos. Veremos cómo las carpetas correspondientes a packages y subpackages han desaparecido.

Existe una forma abreviada de definir estructuras de packages. Cuando empezamos un proyecto es posible que ya hayamos pensado cómo vamos a organizarlo. Podemos definir de forma directa una estructura usando, desde el nivel raíz, la creación de un package con la notación de puntos. BlueJ creará de forma automática las carpetas y subcarpetas necesarias. Por ejemplo, lo mismo que hicimos antes en dos pasos (primero crear el package *entradas* y luego crear el package *usuario*), podemos hacerlo dándole a Edit -> New package, y escribiendo la jerarquía o ruta de packages: en nuestro caso escribimos *entradas.usuarios*. De esta manera, BlueJ nos crea automáticamente los packages anidados y las carpetas y subcarpetas correspondientes.

Veamos un ejemplo de aplicación de lo expuesto hasta ahora. Para nuestro programa `TestPseudoAleatorios` (que lo teníamos funcionando correctamente en el directorio raíz) vamos a suponer que nos llevamos la clase `EntradaDeTeclado` a el package `entradas.usuarios`. Para ello creamos una clase de igual nombre dentro del package, copiamos el código de la clase que estaba en la raíz y luego eliminamos la clase de la raíz. Al crear la clase dentro del package automáticamente nos aparece una línea de cabecera: `package entradas.usuarios;`. Esta línea es la que **codifica a qué package pertenece la clase**. A continuación irá el código tal y como lo habíamos definido:

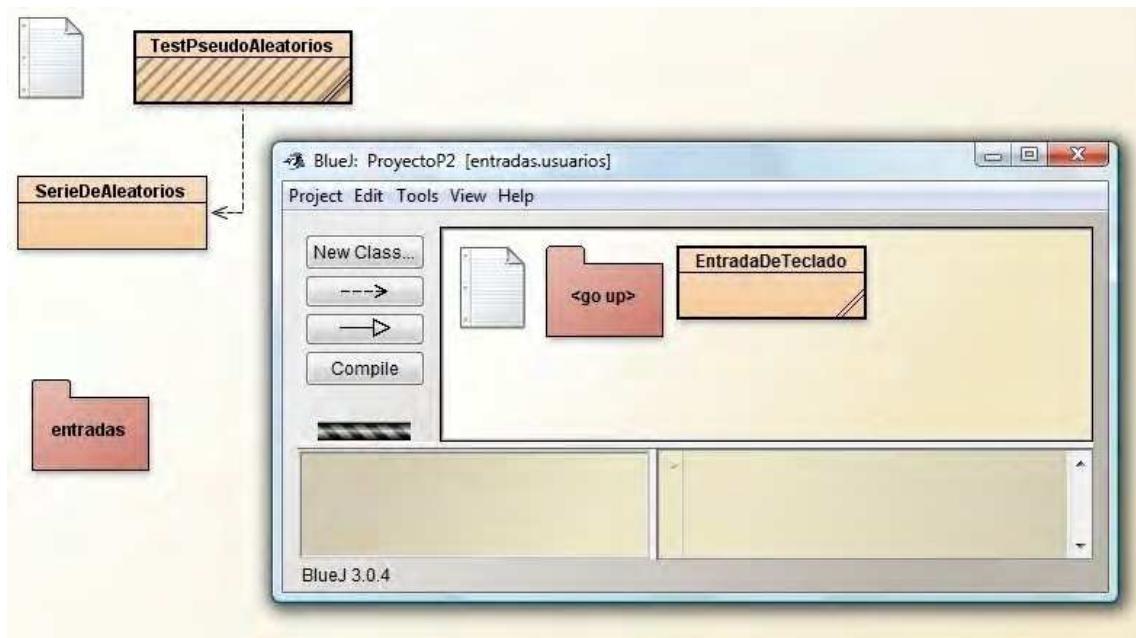
```
package entradas.usuarios; //Define a qué package pertenece la clase
import java.util.Scanner; //Importación de la clase Scanner desde la biblioteca Java
// Ejemplo aprenderaprogramar.com
public class EntradaDeTeclado { // Definimos la clase EntradaDeTeclado
    private String entradaTeclado; //Variable de instancia (campo) del método

    public EntradaDeTeclado () { //Constructor
        entradaTeclado=""; } //Cierre del constructor

    public void pedirEntrada () { //Método de la clase
        Scanner entradaEscaner = new Scanner (System.in);
        entradaTeclado = entradaEscaner.nextLine ();
    } //Cierre del método pedirEntrada

    public String getEntrada () { return entradaTeclado; } //Cierre del método getEntrada
} //Cierre de la clase
```

La visualización de la estructura del proyecto es similar a esta:



Al haber eliminado una clase (que hemos puesto en un package) hay que compilar de nuevo. Y al compilar nos salta el error: `cannot find symbol - class EntradaDeTeclado`. ¿Qué ocurre? Que el que la clase se encuentre en un package **afecta a su visibilidad**. En este caso la clase `TestPseudoAleatorios` no puede “ver” a la clase `EntradaDeTeclado`. Vamos a solucionarlo con una importación a la clase

TestPseudoAleatorios de la clase EntradaDeTeclado con esta sintaxis: `import entradas.usuarios.EntradaDeTeclado;`. También podríamos haber usado `import entradas.usuarios.*;`. En el esquema de clases de BlueJ vemos esto:



TestPseudoAleatorios no aparece conectado con una flecha de relación de uso con el package `entradas`. ¿Por qué? Porque la flecha se reserva para relaciones de uso directo de una clase a otra dentro de un mismo paquete. Cuando se trata de usos a través de `import` no nos aparece de forma explícita la representación de dicha relación.

Una última cuestión: ni los paquetes superiores reconocen las clases de los subpaquetes ni al revés: aquí, en principio, nadie conoce a nadie. Por tanto también hemos de importar las clases o paquetes incluso desde paquetes que están jerárquicamente en la misma línea y por encima de otros paquetes.

EJERCICIO

Crea un proyecto Java con la siguiente estructura:

- Un package denominado `formas` dentro del cual existan los siguientes packages: `formas1dimension`, `formas2dimensiones` y `formas3dimensiones`.
- Dentro del package `formas1dimension` deben existir las clases `Recta` y `Curva`.
- Dentro del package `formas2dimensiones` deben existir las clases `Triangulo`, `Cuadrilatero`, `Elipse`, `Parabola` e `Hiperbola`.
- Dentro del package `formas3dimensiones` deben existir las clases `Cilindro`, `Cono` y `Esfera`.

Nota: crea las clases sin rellenar su código, simplemente para poder visualizar que están dentro de los packages adecuados.

Visualiza el resultado y comprueba que las clases están agrupadas de forma adecuada.

Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00675B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

COPIAR ARRAYS Y COMPARAR ARRAYS. IDENTIDAD E IGUALDAD

Al trabajar con arrays de tipos primitivos o de objetos se nos puede plantear la necesidad de copiar arrays. La copia de arrays está permitida pero conviene ser cauto cuando realicemos procesos de este tipo. Recordar que un array es un objeto (aunque sea especial) y por tanto la variable que lo nombra en realidad contiene un puntero al objeto, no el objeto en sí mismo.



Al hacer una operación del tipo `array1 = array2`, el puntero de `array1` apunta al mismo objeto que `array2` mientras que el objeto al que apuntaba `array1` queda inaccesible. A partir de ese momento existe identidad entre los arrays y la comparación usando `==` nos devolverá `true`. A través de código, vamos a plantearnos distintas situaciones y a comentar cuáles son los resultados.

EJERCICIO EJEMPLO RESUELTO 1. COPIAR ESTABLECIENDO UNA RELACIÓN DE IDENTIDAD ENTRE ARRAYS (aplicable a arrays de tipos primitivos y a arrays de objetos)

```
//Test de copia de arrays aprenderaprogramar.com
public class TestCopiaArrays {

    public static void main (String [ ] Args) {
        int [ ] miArray1 = {2, -4, 3, -7};
        for (int i=0; i<miArray1.length; i++) {
            System.out.print ("miArray1[" + i +"]= " + miArray1[i]+"; ");
        }
        System.out.println("");
        int [ ] otroArray = {1, 2, 4, 8};
        for (int i=0; i<otroArray.length; i++) {
            System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }
        System.out.println("");

        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray) );
        System.out.println("");

        otroArray = miArray1; //otroArray pasa a ser el mismo objeto que miArray1

        for (int i=0; i<otroArray.length; i++) {
            System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }
        System.out.println("");
        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray) );
    } //Cierre del main

} //Cierre de la clase
```

Trata de predecir tú mismo el resultado y compáralo con el que ofrecemos. Resultado:

```
miArray1[0]= 2; miArray1[1]= -4; miArray1[2]= 3; miArray1[3]= -7;
otroArray[0]= 1; otroArray[1]= 2; otroArray[2]= 4; otroArray[3]= 8;
¿Son el mismo objeto? ... false
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7;
¿Son el mismo objeto? ... true
```

EJERCICIO EJEMPLO RESUELTO 2. INTENTO DE COMPARAR ARRAYS (RELACIÓN DE IGUALDAD)

Supongamos que queremos comparar la igualdad entre dos arrays. Una idea (mala idea) podría ser aplicar el método equals directamente sobre el array. Un código de prueba podría ser este:

```
//Test uso incorrecto de equals aprenderaprogramar.com
public class TestUsosIncorrectoEquals {
    public static void main (String [ ] Args) {
        int [ ] miArray1 = {2, -4, 3, -7};
        for (int i=0; i < miArray1.length; i++) {
            System.out.print ("miArray1[" + i +"]= " + miArray1[i]+"; ");
        }
        System.out.println();
        int [ ] otroArray = {2, -4, 3, -7};
        for (int i=0; i < otroArray.length; i++) {
            System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }
        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray) );
        System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + (miArray1.equals(otroArray) ) );
        otroArray = miArray1; //otroArray pasa a ser el mismo objeto que miArray1

        for (int i=0; i<otroArray.length; i++) { System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; "); }

        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray) );
        System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + (miArray1.equals(otroArray) ) );
    } //Cierre del main
} //Cierre de la clase
```

Trata de predecir tú mismo el resultado y compáralo con el que ofrecemos. Resultado:

```
miArray1[0]= 2; miArray1[1]= -4; miArray1[2]= 3; miArray1[3]= -7;
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7; ¿Son el mismo objeto? ... false
¿Tienen el mismo contenido (relación de igualdad)? ... false
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7; ¿Son el mismo objeto? ... true
¿Tienen el mismo contenido (relación de igualdad)? ... true
```

Con este ejemplo comprobamos que el método equals aplicado directamente sobre arrays no funciona ya que en el primer caso teniendo el mismo contenido nos dice que la relación de igualdad es falsa.

¿Por qué? Porque ya dijimos en su momento que los arrays son unos objetos especiales que no tienen una clase que los implemente. Si no tienen clase, no tienen métodos (al menos directamente), y el método que se aplica es el usado por defecto por Java con resultados imprevisibles o no deseados. La solución para comparar arrays es usar otra clase que permite realizar operaciones con arrays. Esta clase es la **clase Arrays**. Una vez más la terminología Java nos puede inducir a confusión: hablamos de array como colección de tipos primitivos u objetos, y al mismo tiempo Arrays (con mayúsculas) es el nombre de una clase Java que sirve para manipular arrays.

EJERCICIO

Crea un programa Java donde declares un array de enteros tipo int miArray1 cuyo contenido inicial sea {2, -4, 3, -7}. Muestra su contenido por pantalla. Copia el contenido de este array a un ArrayList denominado lista1 y muestra su contenido por pantalla. ¿Qué tipo de datos almacena el array? ¿Qué tipo de datos almacena el ArrayList?

Puedes comprobar si tu código y respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00676B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

LA CLASE ARRAYS DEL API DE JAVA. EQUALS, COPYOF, FILL.

En la documentación de la clase Arrays del API de Java podemos encontrar, entre otras cosas, lo siguiente:

`java.util`

Class Arrays

```
java.lang.Object
└ java.util.Arrays
```

Esta clase contiene varios métodos para manipular arrays (por ejemplo para ordenar un array o buscar un valor u objeto dentro de él) y para comparar arrays.



Dado que pertenece al package `util`, para poder usar esta clase habremos de incluir en cabecera `import java.util.Arrays; o bien import java.util.*;`. Al igual que los arrays son unos objetos que hemos dicho son especiales (al carecer de métodos), podemos decir que la clase Arrays es una clase un tanto especial por cuanto carece de constructor. Digamos que directamente al cargar la clase con la sentencia `import` correspondiente automáticamente se crea un objeto denominado Arrays que nos permite realizar manipulaciones con uno o varios arrays (p. ej. ordenar un array, comparar dos arrays, etc.). Dicho objeto podemos utilizarlo directamente: no es necesario declararlo ni crearlo, eso es automático en Java, y por eso decimos que esta clase es una clase especial. La clase Arrays tiene muchos métodos, entre ellos varios métodos `equals` (sobrecarga del método) que hacen que `equals` sea aplicable tanto a arrays de los distintos tipos primitivos como a arrays de objetos. En concreto el método aplicable a arrays de enteros primitivos es:

```
static boolean equals(int[] a, int[] a2)
```

Devuelve true si los dos arrays especificados tienen relación de igualdad entre sí.

Vamos a aplicar este método para comparar el contenido de dos arrays de enteros (relación de igualdad). La aplicación al resto de tipos primitivos y objetos es análoga. La sintaxis en general es:

`Arrays.equals(nombreArray1, nombreArray2).`

EJERCICIO EJEMPLO RESUELTO 3. COMPARAR ARRAYS (RELACIÓN DE IGUALDAD) USANDO LA CLASE ARRAYS

```

import java.util.Arrays;
//Test comparar arrays relación de igualdad aprenderaprogramar.com
public class TestCompararArrays {
    public static void main (String [ ] Args) {
        int [ ] miArray1 = {2, -4, 3, -7};
        for (int i=0; i<miArray1.length; i++) {
            System.out.print ("miArray1[" + i +"]= " + miArray1[i]+"; ");
        }
        System.out.println ("");
        int [ ] otroArray = {2, -4, 3, -7};
        for (int i=0; i<otroArray.length; i++) {
            System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }
        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray));
        System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + Arrays.equals(miArray1, otroArray));
        otroArray = miArray1; //otroArray pasa a ser el mismo objeto que miArray1
        for (int i=0; i<otroArray.length; i++) { System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }
        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray));
        System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + Arrays.equals(miArray1, otroArray));
    } //Cierre del main
} //Cierre de la clase

```

Trata de predecir tú mismo el resultado y compáralo con el que ofrecemos. Resultado:

```

miArray1[0]= 2; miArray1[1]= -4; miArray1[2]= 3; miArray1[3]= -7;
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7; ¿Son el mismo objeto? ... false
¿Tienen el mismo contenido (relación de igualdad)? ... true
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7; ¿Son el mismo objeto? ... true
¿Tienen el mismo contenido (relación de igualdad)? ... true

```

El resultado ahora sí es correcto, porque hemos usado correctamente la clase Arrays para realizar la comparación entre dos arrays.

EJERCICIO EJEMPLO RESUELTO 4. COPIAR CONTENIDOS ENTRE ARRAYS SIN ESTABLECER RELACIÓN DE IDENTIDAD (“manual”, aplicable a tipos primitivos y a objetos).

Antes vimos cómo asignar el contenido de un array a otro haciendo que la variable apunte al mismo objeto. Vamos a ver ahora cómo copiar el contenido entre dos arrays pero manteniendo que cada variable denominadora del array apunte a un objeto diferente:

```

import java.util.Arrays;
//Test copia arrays con igualdad sin identidad aprenderaprogramar.com
public class TestCopiaConIgualdadSinIdentidad {
    public static void main (String [ ] Args) {
        int [ ] miArray1 = {2, -4, 3, -7};
        for (int i=0; i<miArray1.length; i++) {
            System.out.print ("miArray1[" + i +"]= " + miArray1[i]+"; ");
        }
        System.out.println ();
    }
}

```

```

int [ ] otroArray = {1, 2, 4, 8};
for (int i=0; i<otroArray.length; i++) {
    System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
}

System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray ) );
System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + Arrays.equals(miArray1, otroArray) );

//Realizamos una asignación elemento a elemento
for (int i=0; i < otroArray.length; i++) {
    otroArray[i] = miArray1[i];
}
for (int i=0; i < otroArray.length; i++) {
    System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
}

System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray ) );
System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + Arrays.equals(miArray1, otroArray) );
} //Cierre del main
} //Cierre de la clase

```

Trata de predecir tú mismo el resultado y compáralo con el que ofrecemos. Resultado:

```

miArray1[0]= 2; miArray1[1]= -4; miArray1[2]= 3; miArray1[3]= -7;
otroArray[0]= 1; otroArray[1]= 2; otroArray[2]= 4; otroArray[3]= 8; ¿Son el mismo objeto? ... false
¿Tienen el mismo contenido (relación de igualdad)? ... false
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7; ¿Son el mismo objeto? ... false
¿Tienen el mismo contenido (relación de igualdad)? ... true

```

EJERCICIO EJEMPLO RESUELTO 5. COPIAR CONTENIDOS ENTRE ARRAYS SIN ESTABLECER RELACIÓN DE IDENTIDAD (Usando el método copyOf de la clase Arrays, aplicable a tipos primitivos y a objetos).

El método copyOf de la clase Arrays nos permite:

- Copiar un array manteniendo el número de elementos.
- Copiar un array agrandando el número de elementos que tiene, quedando los nuevos elementos llenados con valores cero o nulos.
- Copiar un array empequeñeciendo el número de elementos que tiene; los elementos que no caben en el nuevo array, dado que tiene menor capacidad, se pierden (el array queda truncado).

copyOf es un método sobrecargado. En el caso de arrays de enteros su signatura es la siguiente:

```
static int[ ] copyOf(int[ ] original, int newLength)
```

Copia el array especificado, truncando o llenando con ceros (si fuera necesario) de manera que la copia tenga el tamaño especificado.

Para el resto de tipos primitivos su sintaxis es análoga: `Arrays.copyOf (nombreDelArray, n)` siendo `n` un entero que define la nueva longitud del array (`n` puede ser mayor, menor o igual que la longitud del array original). El código de ejemplo sería este (usamos el `copyOf` sin variar la longitud del array):

```
import java.util.Arrays;
//Test uso de copyOf método clase Arrays aprenderaprogramar.com
public class TestUso_copyOf_1 {
    public static void main (String [ ] Args) {
        int [ ] miArray1 = { 2, -4, 3, -7 };
        for (int i=0; i<miArray1.length; i++) {
            System.out.print ("miArray1[" + i +"]= " + miArray1[i]+"; ");
        }
        System.out.println ("");
        int [ ] otroArray = { 1, 2, 4, 8 };
        for (int i=0; i<otroArray.length; i++) {
            System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }
        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray ) );
        System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + Arrays.equals(miArray1, otroArray) );

        //Copiamos el array utilizando el método copyOf de la clase Arrays
        otroArray = Arrays.copyOf(miArray1, miArray1.length);

        for (int i=0; i<otroArray.length; i++) {
            System.out.print ("otroArray[" + i +"]= " + otroArray[i]+"; ");
        }

        System.out.println ("¿Son el mismo objeto? ... " + (miArray1==otroArray ) );
        System.out.println ("¿Tienen el mismo contenido (relación de igualdad)? ... " + Arrays.equals(miArray1, otroArray) );
    } //Cierre del main
} //Cierre de la clase
```

Trata de predecir tú mismo el resultado y compáralo con el que ofrecemos. Resultado:

```
miArray1[0]= 2; miArray1[1]= -4; miArray1[2]= 3; miArray1[3]= -7;
otroArray[0]= 1; otroArray[1]= 2; otroArray[2]= 4; otroArray[3]= 8; ¿Son el mismo objeto? ... false
¿Tienen el mismo contenido (relación de igualdad)? ... false
otroArray[0]= 2; otroArray[1]= -4; otroArray[2]= 3; otroArray[3]= -7; ¿Son el mismo objeto? ... false
¿Tienen el mismo contenido (relación de igualdad)? ... true
```

Hemos comprobado que **el método `copyOf` de la clase `Arrays` realiza una copia elemento a elemento** entre los contenidos de dos arrays pero no hace que los punteros apunten al mismo objeto. Prueba a variar la longitud que se le pasa como parámetro al método `copyOf`, por ejemplo:

```
otroArray = Arrays.copyOf(miArray1, miArray1.length+2); //Resto del código igual
```

```
otroArray = Arrays.copyOf(miArray1, miArray1.length-2); //Resto del código igual
```

Comprueba que los resultados son el alargamiento del array y su relleno con ceros, o el acortamiento con pérdida de los datos (truncamiento) que no caben debido al recorte de la longitud. En el caso de alargamiento o expansión del array cuando se trata de un array que no sea de enteros, si son tipos numéricos se llenan los excedentes con ceros, si son booleanos se llenan los excedentes con false, si son char se llenan de caracteres vacío, y si son objeto se llenan los excedentes con null.

RELENAR UN ARRAY CON UN VALOR U OBJETO. MÉTODO FILL DE LA CLASE ARRAYS

La clase Arrays tiene un método, denominado fill, sobrecargado, que permite llenar un array con un determinado valor u objeto. En el caso de arrays de enteros la signatura es:

```
static void fill(int[ ] a, int val)
```

Asigna el valor entero especificado a cada elemento del array de enteros indicado.

En general la sintaxis será: `Arrays.fill (nombreDelArray, valor con el que se rellena)`. El valor con el que se rellena depende del tipo del array. Por ejemplo, si es un array de tipo booleano, tendremos que rellenarlo bien con `true` o bien con `false`, no podremos rellenarlo con un tipo que no sea coherente. Ejemplos de uso:

`Arrays.fill (resultado, '9');` Como rellenamos con un carácter, resultado habrá de ser un array de caracteres, ya que en caso contrario no habría coincidencia de tipos.

`Arrays.fill (permitido, true);` Como rellenamos con un `true`, resultado será un array de booleanos. De otra manera, no habría coincidencia de tipos. Ejemplo de código:

```
import java.util.Arrays;
public class TestMetodoFillArrays {
    public static void main (String [ ] Args) { //main cuerpo del programa ejemplo aprenderaprogramar.com
        int [ ] miArray = new int[10];
        Arrays.fill(miArray, 33);
        for (int tmp: miArray) { System.out.print (tmp + ","); } //Recorrido del array con un for each
    } //Cierre del main y de la clase
```

Ejecuta el código y comprueba que el resultado es: 33,33,33,33,33,33,33,33,33,. Es decir, el array queda lleno en todos sus elementos con 33.

En caso de que el array tenga contenidos previos al aplicarle el fill, todos sus elementos quedarán reemplazados por el elemento de relleno. No obstante, hay otro método que permite especificar los índices de relleno de modo que se pueda preservar parte del contenido previo del array:

```
static void fill(int[ ] a, int fromIndex, int toIndex, int val)
```

Asigna el valor entero especificado a cada elemento del rango indicado para el array especificado.

Escribe un fragmento de código utilizando esta signatura del método fill y comprueba sus resultados.

Próxima entrega: CU00677B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONCEPTO DE INTERFAZ O INTERFACE JAVA. AMPLIACIÓN.

Ya hemos dicho que interface en Java es una palabra que puede tener diferentes significados. Vamos a repasar algunos significados que ya hemos visto y a introducir otros nuevos. Es habitual que entre las personas que estudian Java haya dificultades para entender el concepto de interface. Esto es hasta cierto punto normal porque dicho término tiene distintos usos.



Ya hemos dicho que interface en Java es una palabra que puede tener diferentes significados. Vamos a repasar algunos significados que ya hemos visto y a introducir otros nuevos.

- a) **Interface:** parte visible y pública de una clase que describe qué hace y cómo usarla. La documentación de una clase en el API de Java vendría siendo su interface.
- b) **Interface:** parte visible y pública de un método que describe qué hace y cómo usarlo (signatura del método + instrucciones de uso). La documentación de un método en el API de Java vendría siendo su interface.
- c) **Interfaz Gráfica de Usuario,** interfaz de usuario o GUI (Graphical User Interface): es el entorno de objetos gráficos disponibles para un usuario en el marco de una aplicación o sistema operativo. El sistema operativo MS-Dos se basaba en intérpretes de comando (escritura de instrucciones por consola) pero Windows se basa en una interfaz gráfica de usuario (su entorno de escritorio), Linux en otra y Macintosh en otra.
- d) **Herramientas para crear Interfaces gráficas de usuario** en Java. Hacemos referencia principalmente a los paquetes (packages) del API de Java *swing* y *awt* (Abstract Windowing Toolkit). Las clases de estos paquetes permiten crear interfaces gráficas de usuario basadas en ventanas estilo “Windows” para nuestras aplicaciones.
- e) **Interfaces de Java:** son unas entidades abstractas conceptualmente por encima de las clases cuyo concepto vamos a introducir a continuación.

Para explicar el concepto de interface nos valdremos de un ejemplo. Supongamos que al crear un programa creamos una ciudad. Para crear la ciudad no partimos de cero: disponemos de edificios prefabricados (las clases del API de Java). Pero también disponemos de algo más: de normas de

urbanismo ya definidas (las interfaces del API de Java). Cada norma vamos a decir que es una interface: nos dice qué debemos cumplir para que al construir un edificio (clase) se pueda calificar con un nombre determinado. Supongamos una norma denominada “Edificio a dos aguas”, cuyo contenido incluye:

- a) El edificio habrá de tener cuatro paredes.
- b) El edificio habrá de tener un tejado formado por dos planos.
- c) Otras especificaciones.

Si al construir un edificio se cumplen las condiciones de las normas de urbanismo, el ayuntamiento nos permite que en la publicidad y escrituras del edificio conste que se trata de un edificio a dos aguas. Si no cumplimos las especificaciones, no podemos usar esa denominación. Por ejemplo, estaría prohibido que un edificio con forma de pentágono y cinco paredes se denominara “Edificio a dos aguas”. Por el contrario, sería posible denominar edificio a dos aguas a una capilla que cumpla la norma, también a una vivienda unifamiliar que la cumpla, o a una biblioteca que la cumpla. Trasladémoslo a Java. Consideremos la interface “List” que equivale a nuestra norma, cuyo contenido incluye:

- a) La clase habrá de tener un método cuya firma sea `get(int index)` que devuelva el objeto de la lista en la posición especificada por el entero `index`.
- b) La clase habrá de tener un método cuya firma sea `isEmpty()` que devuelva un valor booleano `true` si la lista no contiene objetos o `false` en caso contrario.
- c) Otras especificaciones.

Si una clase cumple y declara que cumple las especificaciones de la interfaz, decimos que esa clase implementa la interfaz. Si no cumple las especificaciones, no la implementa. Para implementar la interfaz la clase ha de incluir todos los métodos que defina la interfaz. Por ejemplo, una clase que no incluya el método `get(int index)` no cumplirá la especificación y por tanto no podrá decirse que implemente la interfaz List. Veamos ahora cómo empieza la documentación de esta interface del API de Java:

java.util

Interface List<E>

All Superinterfaces:

[Collection<E>](#), [Iterable<E>](#)

All Known Implementing Classes:

[AbstractList](#), [AbstractSequentialList](#), [ArrayList](#), [AttributeList](#), [LinkedList](#), [RoleList](#), [Stack](#), [Vector](#) y otras

Una interface puede tener “normas” superiores, a las que denominamos “**Superinterfaces**”, y “normas” inferiores, a las que denominamos “**Subinterfaces**”, de acuerdo con una jerarquía. A su vez, una interface puede tener clases que la implementan. Por ejemplo en el caso de List, esta interface está implementada por las clases ArrayList, LinkedList, Stack y otras. Todas las clases que implementan una interface podemos decir que tienen algo en común. De esta manera, todas las clases del API de Java están organizadas.

EJERCICIO

Busca información en la documentación oficial de Java sobre la interface Iterable. ¿Qué clases de las que conoces implementan esta interface? ¿Qué método han de tener disponible las clases que implementan esta interface? ¿Para qué sirve este método?

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00678B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONCEPTO DE POLIMORFISMO EN JAVA. ¿QUÉ ES?

En el anterior apartado del curso hemos indicado que es posible denominar “edificio a dos aguas” a una capilla que cumpla la norma, también a una vivienda unifamiliar que la cumpla, o a una biblioteca que la cumpla. Y que podemos decir que son List las clases que cumplen con la interface como ArrayList, LinkedList y otras.



Fijémonos ahora en estas formas de sintaxis:

FORMA 1	FORMA 2
<pre>private ArrayList <Integer> serieAleatoria; ... ejemplo aprenderaprogramar.com serieAleatoria = new ArrayList <Integer> ();</pre>	<pre>private List <Integer> serieAleatoria; ... ejemplo aprenderaprogramar.com serieAleatoria = new ArrayList<Integer> ();</pre>
Aquí se construirá un edificio a dos aguas tipo capilla; ... ejemplo aprenderaprogramar.com Construimos un edificio a dos aguas tipo capilla;	Aquí se construirá un edificio a dos aguas; ... ejemplo aprenderaprogramar.com Construimos un edificio a dos aguas tipo capilla;

Lo que estamos comprobando aquí es que Java permite declarar variables sin especificar la clase concreta a la que van a pertenecer, sino únicamente diciendo qué interface van a cumplir. Por este motivo decimos que **una interface define un tipo**. Esto explica a su vez por qué en la documentación del API de Java nos encontramos ocasiones en que vemos que un método requiere como parámetro un tipo que no se corresponde con una clase, sino con una interface. Supongamos que un método pide como parámetro un tipo List. Eso significa que admite que se le pase como parámetro tanto un ArrayList como un LinkedList como otro objeto que cumpla con la interface. En este caso diríamos que el parámetro es polimórfico porque admite distintas formas de objeto, no solo una. El polimorfismo se manifiesta de distintas formas en Java como iremos viendo en los próximos apartados. Trata de razonar sobre el significado del siguiente código:

```
//Ejemplo aprenderaprogramar.com
ArrayList <List> misListas = new ArrayList <List> ();
ArrayList<Integer> miListaIntegers = new ArrayList<Integer>();
LinkedList<String> miListaStrings = new LinkedList<String>();
misListas.add (miListaIntegers);
misListas.add (miListaStrings);
```

Con los conocimientos explicados hasta el momento debemos ser capaces de interpretar estas líneas. En primer lugar declaramos un ArrayList que va a contener objetos de tipo List: eso significa que dentro del ArrayList podrá haber ArrayList, LinkedList, Stack, etc. A continuación creamos un ArrayList de integers y un LinkedList de strings. Y finalmente, añadimos esos objetos a misListas. Esta variable decimos que es polimórfica porque contiene objetos que no son solo de un tipo, sino de varios tipos.

EJERCICIO

La interface Set de Java es implementada por las clases HashSet y TreeSet. Busca información sobre estas clases en la documentación del api Java. Crea un programa Java que haga lo siguiente:

- a) Declarar un ArrayList de objetos de tipo Set
- b) Crear un objeto de tipo HashSet para contener Strings y haz que contenga las cadenas “sol”, “luna”, “saturno”.
- c) Crear un objeto TreeSet para contener Integers y haz que contenga los números 2, 8, 5.
- d) Añade los objetos HashSet y TreeSet como elementos del ArrayList.
- e) Usa iteradores para recorrer los elementos del ArrayList y recorrer el contenido de cada uno de los elementos y mostrar este contenido por pantalla. Por pantalla deberás obtener “sol”, “luna”, “saturno”, 8, 5, 2.

Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00679B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

TRANSFORMAR ARRAY EN UNA LISTA. MÉTODO ASLIST.

La clase Arrays tiene un método denominado `asList` que devuelve un tipo `List` cuando se le invoca pasándole un array como parámetro. Vamos a plantear el siguiente ejercicio: declararemos un array de `Strings` que contenga los nombres de cuatro animales, y a continuación crearemos un `ArrayList` que tendrá el mismo contenido que el array.



Escribe y compila el siguiente código:

```
import java.util.List; //Ejemplo generar una lista a partir de un array aprenderaprogramar.com
import java.util.ArrayList;
import java.util.Arrays;

public class TestAsList {

    public static void main (String [ ] Args) {
        List <String> miListaDePalabras = new ArrayList <String> ();
        String [ ] miArrayDePalabras = {"Oso", "León", "Buey", "Guepardo"};
        System.out.println ("Contenido inicial del ArrayList: " + miListaDePalabras);
        miListaDePalabras = Arrays.asList (miArrayDePalabras);
        System.out.println ("Contenido actual del ArrayList: " + miListaDePalabras + "\n");

        List <Integer> miListaDeEnteros = new ArrayList <Integer> ();
        Integer [ ] miArrayDeEnteros = {11, -34, 56, 78};
        System.out.println ("Contenido inicial del ArrayList: " + miListaDeEnteros);
        miListaDeEnteros = Arrays.asList (miArrayDeEnteros);
        System.out.println ("Contenido actual del ArrayList: " + miListaDeEnteros);
    } //Cierre del main
} //Cierre de la clase
```

El resultado es similar a este:

```
Contenido inicial del ArrayList: []
Contenido actual del ArrayList: [Oso, León, Buey, Guepardo]
Contenido inicial del ArrayList: []
Contenido actual del ArrayList: [11, -34, 56, 78]
```

Fíjate que hemos definido un array de `Integers` y no un array de `int`. Si utilizáramos una declaración como `int [] miArrayDeEnteros = {11, -34, 56, 78};` tendríamos un error de compatibilidad de tipos. Este es un primer ejemplo de uso de un tipo (`List`) definido por una interface. Vamos a ver ahora otra forma

de uso de un tipo definido por una interface. Para ello vamos a consultar la documentación de la clase `ArrayList`, y en concreto uno de sus constructores:

[ArrayList\(Collection<? extends E> c\)](#)

Crea un `ArrayList` que contiene los elementos de la colección que se pase como parámetro, en el orden que devuelve el iterador de la colección.

Este es otro constructor de la clase `ArrayList`, distinto al que estamos acostumbrados a usar basado en crear un `ArrayList` vacío. En este código vemos un ejemplo de uso:

```
import java.util.ArrayList; import java.util.LinkedList; //Ejemplo aprenderaprogramar.com
public class TestConstructor2AL {
    public static void main (String [ ] Args) {
        LinkedList<String> miListaStrings = new LinkedList<String>();
        miListaStrings.add ("Liebre");
        miListaStrings.add ("Perro");
        ArrayList <String> miArrayListStrings = new ArrayList <String> (miListaStrings); //Ejemplo nuevo constructor
        System.out.println ("Contenido del LinkedList " + miListaStrings);
        System.out.println ("Contenido del ArrayList " + miArrayListStrings);
    } } //Cierre del main y de la clase
```

En este ejemplo vemos cómo creamos un `ArrayList` pasando como parámetro un tipo `Collection`. `LinkedList` es una de las clases que implementa la interface `Collection`, por tanto podemos pasarla en este constructor. De esta forma, el `ArrayList` tiene el contenido inicial que tiene la colección que se le pasa como parámetro y al ejecutar el programa obtenemos: Contenido del LinkedList [Liebre, Perro] ; Contenido del ArrayList [Liebre, Perro].

EJERCICIO

Declara un array que contenga seis booleanos que sean true, true, false, false, true, false. A continuación, crea una lista de tipo `LinkedList` a partir de dicho array usando la clase `Arrays` del api de Java. Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00680B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

DOCUMENTAR PROYECTOS JAVA CON JAVADOC

Documentar un proyecto es algo fundamental de cara a su futuro mantenimiento. Cuando programamos una clase, debemos generar documentación lo suficientemente detallada sobre ella como para que otros programadores sean capaces de usarla sólo con su interfaz. No debe existir necesidad de leer o estudiar su implementación, lo mismo que nosotros para usar una clase del API Java no leemos ni estudiamos su código fuente.



Javadoc es una utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java. **Javadoc es el estándar para documentar clases de Java.** La mayoría de los IDEs utilizan javadoc para generar de forma automática documentación de clases. BlueJ también utiliza javadoc y permite la generación automática de documentación, y visionarla bien de forma completa para todas las clases de un proyecto, o bien de forma particular para una de las clases de un proyecto.

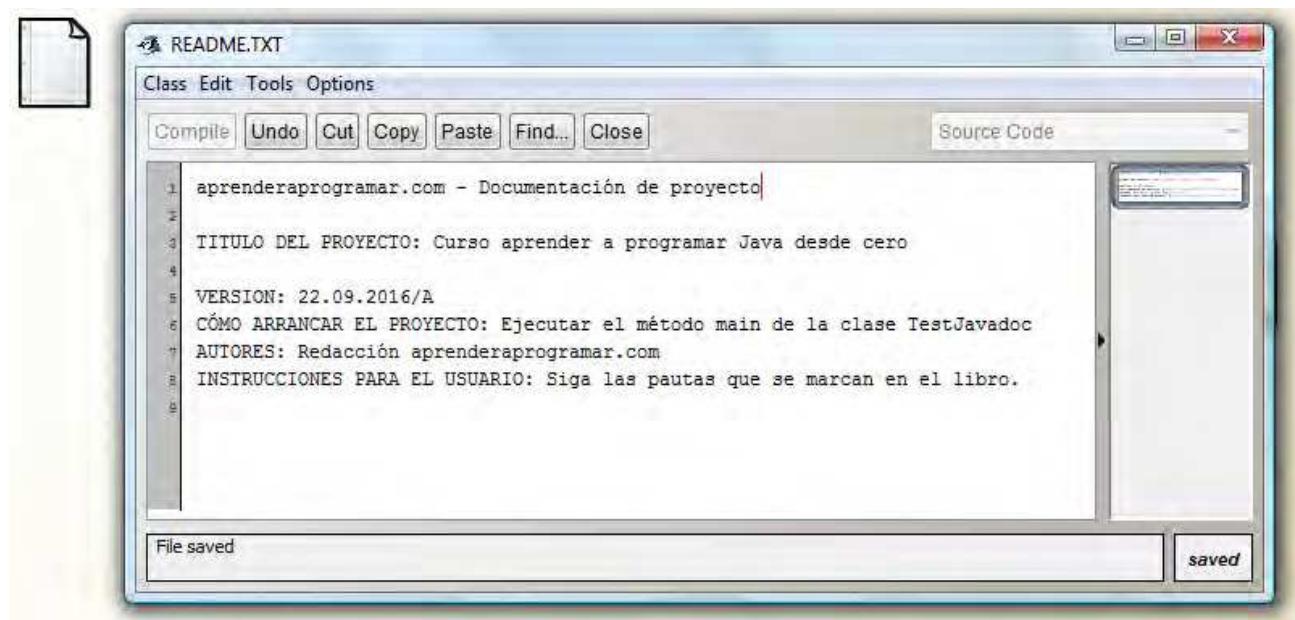
Veamos en primer lugar qué se debe incluir al documentar una clase:

- a) **Nombre de la clase, descripción general, número de versión, nombre de autores.**
- b) **Documentación de cada constructor o método** (especialmente los públicos) incluyendo: nombre del constructor o método, tipo de retorno, nombres y tipos de parámetros si los hay, descripción general, descripción de parámetros (si los hay), descripción del valor que devuelve.

Las variables de instancia o de clase no se suelen documentar a nivel de javadoc.

Para generar la documentación de un proyecto automáticamente hemos de seguir unas normas a la hora de realizar los comentarios dentro del mismo. Si las hemos seguido, en BlueJ disponemos de la opción del menú Tools --> Project Documentation, que nos abre la documentación del proyecto en un navegador web. Para ver la documentación de una clase específica en BlueJ, debemos abrir la ventana de código, y en la parte superior derecha cambiar la pestaña que pone "Source code" por la opción "Documentation".

Además de documentar las clases, **todo proyecto debería tener un archivo Leeme o Readme.** En BlueJ podemos acceder al readme.txt de proyecto haciendo doble click en el ícono que representa una hoja en la parte superior izquierda del diagrama de clases. En el readme.txt sería adecuado incluir al menos: título del proyecto, descripción, versión, cómo arrancar el proyecto, autores e instrucciones para los usuarios. Haz doble click en el ícono e introduce una descripción para tu proyecto.



Una vez cierres la ventana, la información incluida se guarda automáticamente. Para que javadoc sea capaz de generar documentación automáticamente han de seguirse estas reglas:

- a) La documentación para javadoc ha de incluirse entre símbolos de comentario que han de empezar con una barra y doble asterisco, y terminar con un asterisco y barra simple.

```
/**  
 * Esto es un comentario para javadoc ejemplo aprenderaprogramar.com  
 */
```

- b) La ubicación le define a javadoc qué representa el comentario: si está incluido justo antes de la declaración de clase se considerará un comentario de clase, y si está incluido justo antes de la firma de un constructor o método se considerará un comentario de ese constructor o método.
- c) Para alimentar javadoc se usan ciertas palabras reservadas (tags) precedidas por el carácter "@", dentro de los símbolos de comentario javadoc. Si no existe al menos una línea que comience con @ no se reconocerá el comentario para la documentación de la clase.

En la tabla siguiente mostramos algunas de las palabras reservadas (tags), aunque hay más de las que aquí incluimos. Si necesitas una lista completa de las tags con su correspondiente uso realiza una búsqueda de la cadena "javadoc tags" en bing, google o yahoo.

TAG	DESCRIPCIÓN	COMPRENDE
@author	Nombre del desarrollador.	Nombre autor o autores
@deprecated	Indica que el método o clase es obsoleto (propio de versiones anteriores) y que no se recomienda su uso.	Descripción
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	Nombre de parámetro y descripción
@return	Informa de lo que devuelve el método, no se aplica en constructores o métodos "void".	Descripción del valor de retorno
@see	Asocia con otro método o clase.	Referencia cruzada referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).
@version	Versión del método o clase.	Versión

Las etiquetas `@author` y `@version` se usan para documentar clases e interfaces. Por tanto no son válidas en cabecera de constructores ni métodos. La etiqueta `@param` se usa para documentar constructores y métodos. La etiqueta `@return` se usa solo en métodos de tipo función.

Dentro de los comentarios se admiten etiquetas HTML, por ejemplo con `@see` se puede referenciar una página web como link para recomendar su visita de cara a ampliar información.

A continuación escribe la documentación javadoc para una clase que hayas desarrollado previamente, tratando de utilizar los tags que hemos visto. Por ejemplo:

```
import java.util.ArrayList;
import java.util.Random;

/**
 * Esta clase define objetos que contienen tantos enteros aleatorios entre 0 y 1000 como se le definen al crear un objeto
 * @author: Mario R. Rancel
 * @version: 22/09/2016/A
 * @see <a href = "http://www.aprenderaprogramar.com" /> aprenderaprogramar.com – Didáctica en programación </a>
 */

public class SerieDeAleatoriosD {

    //Campos de la clase
    private ArrayList<Integer> serieAleatoria;
    /**
     * Constructor para la serie de números aleatorios
     * @param numerolitems El parámetro numerolitems define el número de elementos que va a tener la serie aleatoria
     */
    public SerieDeAleatoriosD (int numerolitems) {
        serieAleatoria = new ArrayList<Integer> ();
        for (int i=0; i<numerolitems; i++) { serieAleatoria.add(0); }
        System.out.println ("Serie inicializada. El número de elementos en la serie es: " + getNumerolitems() );
    } //Cierre del constructor
}
```

```

/**
 * Método que devuelve el número de ítems (números aleatorios) existentes en la serie
 * @return El número de ítems (números aleatorios) de que consta la serie
 */
public int getNumeroItems() { return serieAleatoria.size(); }

/**
 * Método que genera la serie de números aleatorios
 */
public void generarSerieDeAleatorios () {
    Random numAleatorio;
    numAleatorio = new Random ();
    for (int i=0; i < serieAleatoria.size(); i++) { serieAleatoria.set(i, numAleatorio.nextInt(1000)); }
    System.out.print ("Serie generada! ");
}
//Cierre del método
} //Cierre de la clase y del ejemplo aprenderaprogramar.com

```

Ahora, desde la ventana del editor de código de BlueJ (parte superior derecha) elige la opción “Documentation” en vez de “Source Code”. Se te mostrará una vista similar a esta:

Class SerieDeAleatoriosD

[java.lang.Object](#)

└ [SerieDeAleatoriosD](#)

public class [SerieDeAleatoriosD](#) extends java.lang.Object

Esta clase define objetos que contienen tantos enteros aleatorios entre 0 y 1000 como se le definen al crear un objeto de esta clase

See Also: [aprenderaprogramar.com – Didáctica en programación](#)

Constructor Summary

[SerieDeAleatoriosD](#)(int numeroItems)

Constructor para la serie de números aleatorios

Method Summary

void	generarSerieDeAleatorios()
------	--

Método que genera la serie de números aleatorios

int	getNumeroItems()
-----	----------------------------------

Método que devuelve el número de ítems (números aleatorios) existentes en la serie

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait
--

Constructor Detail

SerieDeAleatoriosD

```
public SerieDeAleatoriosD(int numeroItems)
    Constructor para la serie de números aleatorios
```

Parameters:

numeroItems - El parámetro numeroItems define el número de elementos que va a tener la serie aleatoria

Method Detail

generarSerieDeAleatorios

```
public void generarSerieDeAleatorios()
    Método que genera la serie de números aleatorios
```

getNumeroItems

```
public int getNumeroItems()
    Método que devuelve el número de ítems (números aleatorios) existentes en la serie
```

Returns:

El número de ítems (números aleatorios) de que consta la serie

Aunque parte de la información que nos aparece todavía no somos capaces de interpretarla, hemos visto cómo los comentarios introducidos en el código siguiendo unas reglas son transformados en documentación por la herramienta javadoc. La documentación del API de Java está generada de esta manera, a gran escala y con un número de tags mayor que el que hemos empleado nosotros. El API de Java nos resulta útil porque está bien documentado. De cara al mantenimiento, ampliación y conexión de nuestros programas **es fundamental que documentemos el código correctamente** y de forma amplia. La documentación javadoc es imprescindible en un proyecto profesional Java. Esta documentación es ideal que se complementa con tantos comentarios libres adicionales como sean necesarios para interpretar bien el código. Los comentarios que no se crean siguiendo las normas de javadoc no aparecerán en la documentación, pero serán útiles para otros programadores o para nosotros mismos cuando tengamos que consultar el código pasado un tiempo después de haberlo generado.

En este curso no mantenemos la estandarización javadoc por motivos de espacio, pero debes tenerla presente y aplicarla para el desarrollo de cualquier proyecto de índole profesional o académica.

EJERCICIO

Crea una clase denominada miniCalculadoraEjemplo que tenga dos métodos (basados en el uso de métodos de la clase Math): un método valorAbsoluto que recibe un número de tipo double y devuelva su valor absoluto, y otro método raizCuadrada que reciba un número de tipo double y devuelva su raíz cuadrada. Documenta esta clase conforme a los estándares JavaDoc y comprueba cómo se visualiza la documentación. Para comprobar si tu solución es correcta puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00681B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

TIPOS ENUMERADOS (ENUM) EN JAVA

Un tipo enumerado en Java es, de partida, similar al de otros lenguajes de programación. No obstante, incluye una serie de características que lo hacen muy potente. Un tipo enumerado restringe los posibles valores que puede tomar una variable. Esto ayuda a reducir los errores en el código y permite algunos usos especiales interesantes.



Los tipos enumerados se declaran con esta sintaxis tipo:

```
Tipo Público/Privado enum nombreTipoEnum { ELEMENTO1, ELEMENTO2, ELEMENTO3, ..., ELEMENTOn };
```

Dentro de las llaves se declaran las variables (objetos) de que consta el tipo enumerado. Por convención, sus nombres se escriben en letras mayúsculas para recordarnos que son valores fijos (que en cierto modo podemos ver como constantes). Una vez declarado el tipo enumerado, todavía no existen variables hasta que no las creamos explícitamente, de la misma manera que ocurre con cualquier tipo Java. Para crear una variable de tipo enumerado lo haremos con una declaración simple que recuerda a la creación de una variable tipo primitivo: *nombreTipoEnum miNombreElegido*;.

Esta forma de creación de variables de tipo enumerado se justifica porque los tipos enumerados en principio no tienen constructores (más adelante veremos que sí los pueden tener). Los valores de un tipo enumerado son objetos propiamente dichos, ¿de qué tipo? Del tipo enumerado cuyo nombre se haya indicado. No se pueden crear más objetos variantes del tipo enumerado que los especificados en su declaración. Tener en cuenta, porque es una confusión habitual, que los tipos enumerados no son enteros, ni cadenas (aunque a veces podamos hacer que se comporten de forma similar a como lo haría un entero o una cadena). Cada elemento de un enumerado es un objeto único disponible para su uso.

La identificación de cada objeto del tipo se hace con la sintaxis del punto, es decir, nos referimos a un elemento concreto como *nombreDelTipoEnum.ELEMENTO1*. Esta sintaxis nos quiere recordar lo que sería un campo de una clase pero no es así: en este caso es un objeto de un tipo enumerado.

Un tipo enumerado puede ser declarado dentro o fuera de una clase, pero no dentro de un método. Por tanto no podemos declarar un enum dentro de un método main (programa principal); si lo hacemos nos saltará el error de compilación “*enum types must not be local*” (los tipos enumerados no deben ser locales a un método). Por tanto el tipo enumerado lo declararemos o bien antes del *public class...* o bien después del *public class...* pero fuera de cualquier método o constructor. Veamos un primer ejemplo de uso:

```
/*
 * Descripción para javadoc antes de la clase. Clase de ejemplo aprenderaprogramar.com para tipos enumerados
 * @author Alex Rodríguez
 */

public class TestEnum {
    enum TipoDeMadera { ROBLE, CAOBA, NOGAL, CEREZO, BOJ };
    public static void main (String[ ] Args) {
        TipoDeMadera maderaUsuario;
        maderaUsuario = TipoDeMadera.ROBLE;
        System.out.println ("La madera elegida por el usuario es " + maderaUsuario.toString().toLowerCase() );
        System.out.println ("¿Es la madera elegida por el usuario caoba? Resultado: " + (maderaUsuario==TipoDeMadera.CAOBA) );
        System.out.println ("¿Es la madera elegida por el usuario roble? Resultado: " + (maderaUsuario==TipoDeMadera.ROBLE) );
    } //Cierre del main
} //Cierre de la clase
```

Ejecuta el anterior código. El resultado de la ejecución será similar a este:

```
La madera elegida por el usuario es roble
¿Es la madera elegida por el usuario caoba? Resultado: false
¿Es la madera elegida por el usuario roble? Resultado: true
```

Hemos usado el método `toString()` que es un método disponible para la mayoría de las clases en Java y utilizado para representar un objeto en una cadena de texto. Ten en cuenta que usamos este método porque el enumerado no es texto.

El enumerado lo hemos incluido como código dentro de la clase. En realidad, podríamos haberlo dispuesto como una clase independiente que llevara únicamente este código:

```
public enum TipoDeMadera { ROBLE, CAOBA, NOGAL, CEREZO, BOJ }
```

EJERCICIO

Crea una clase `Vehiculo` donde definas un tipo enumerado `MarcaDeVehiculo` cuyos valores posibles serán FORD, TOYOTA, SUZUKI, RENAULT, SEAT. Establece como atributos de la clase matricula (tipo String) y marca (tipo `MarcaDeVehiculo`) y crea los métodos getters (para obtener) y setters (para establecer el valor) de los atributos. Escribe un pequeño programa de prueba donde verifiques que los métodos funcionan correctamente. Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00682B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188



APRENDERAPROGRAMAR.COM

ENUMERADOS COMO
CLASES ENUM EN JAVA.
MÉTODO VALUES.
EJERCICIO EJEMPLO
RESUELTO. (CU00682B)

Sección: Cursos

Categoría: Curso “Aprender programación Java desde cero”

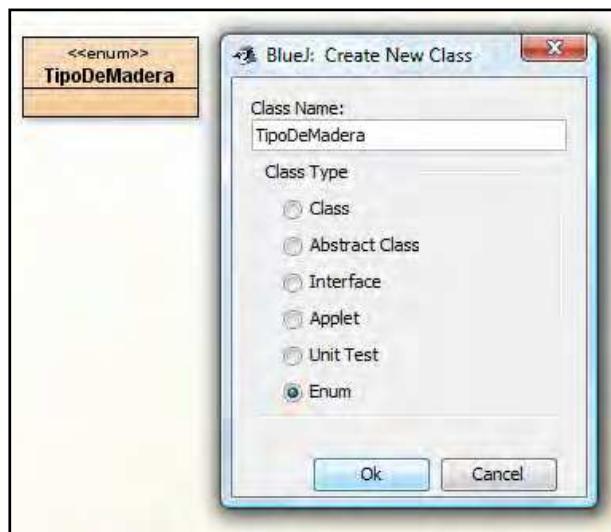
Fecha revisión: 2029

Resumen: Entrega nº82 curso Aprender programación Java desde cero.

Autor: Alex Rodríguez

ENUMERADOS CLASES CON CAMPOS Y CONSTRUCTORES. MÉTODOS VALUES.

Un tipo enumerado en Java es un tipo “especial” que en cierta medida puede usarse como una clase y admite ciertas posibilidades especiales. En BlueJ, cuando pulsamos sobre *New Class...* una opción es *Enum*. Si elegimos esta opción (o si el código que escribimos se corresponde con una clase *Enum*), la clase se verá en el diagrama de clases con la palabra <>Enum>> en su ícono.



De esta manera podemos decir que ya conocemos tres tipos de clases: clases que definen tipos (clases “normales”), clases que incorporan un método *main* (clases tipo “ejecutable”) y por último las clases “enum” que incorporan enumerados y que tienen ciertas peculiaridades. Entre las cosas especiales que se pueden hacer con una clase de tipo *Enum* tenemos:

- a) Se dispone automáticamente de métodos especiales como **el método values()**, que devuelve un array con todos los valores del enum (esto realmente es intrínseco al enumerado, no hace falta que el enumerado vaya en una clase independiente para disponer de esta posibilidad).
- b) Un tipo enumerado puede añadir campos constantes al objeto enumerado y recuperar esos campos. Para ello se usa un constructor especial para enumerados.

En relación al método *values()*, se trata de un método especial que el compilador agrega automáticamente cuando se crea un enum. Este método devuelve un array contenido todos los valores del enumerado en el orden en que son declarados y se usa comúnmente en combinación con el ciclo *for-each* para iterar sobre los valores de un tipo enum. Ejemplo de código. Compílalo (hay que definir el tipo enumerado en una clase independiente previamente) y comprueba el resultado de ejecución:

```
//Ejemplo aprenderaprogramar.com
public class TestEnum2 {
    public static void main (String[ ] Args) {
        TipoDeMadera miTipoDeMadera = TipoDeMadera.CAOBA; //El tipo lo definimos en otra clase como public enum
        System.out.println ("Los posibles tipos de madera son: ");
        for (TipoDeMadera tmp: miTipoDeMadera.values() ) {
            System.out.print(tmp.toString()+"\t"); } //Ejemplo de uso de print con tabulador \t
    } //Cierre del main
} //Cierre de la clase
```

Los posibles tipos de madera son:

ROBLE CAOBA NOGAL CEREZO BOJ

Pasemos a otra cuestión. Al declarar el tipo podemos asociarle campos directamente escribiéndolos entre paréntesis después del nombre del enumerado. Luego declararemos un constructor (sin la palabra clave *public*, ya que el constructor no puede ser público porque no podemos crear objetos del tipo enumerado con la sentencia *new*, pero sí podemos hacer que un objeto lleve asociados determinados parámetros constantes). Veamos un ejemplo que consta de dos clases:

- a) **Clase de tipo Enum:** contiene la definición de los enumerados, de los campos asociados a cada enumerado y los tipos y valores de estos campos, constructor intrínseco y métodos para acceder a los campos.
- b) **Clase que contiene un método main** que usa la clase de tipo Enum y sus métodos.

```
// Ejemplo aprenderaprogramar.com
// Clase que contiene los tipos de madera que usa la empresa, su color y su peso específico en kg/m3
public enum TipoDeMadera {
    ROBLE ("Castaño verdoso", 800), //Separamos con comas
    CAOBA ("Marrón oscuro", 770),
    NOGAL("Marrón rojizo", 820),
    CEREZO ("Marrón cereza", 790),
    BOJ ("Marrón negruzco", 675); //Cuando terminamos cerramos con ;

    //Campos tipo constante
    private final String color; //Color de la madera
    private final int pesoEspecifico; //Peso específico de la madera

    /**
     * Constructor. Al asignarle uno de los valores posibles a una variable del tipo enumerado el constructor asigna
     * automáticamente valores de los campos
     */
    TipoDeMadera (String color, int pesoEspecifico) {
        this.color = color;
        this.pesoEspecifico = pesoEspecifico;
    } //Cierre del constructor

    //Métodos de la clase tipo Enum
    public String getColor() { return color; }
    public int getPesoEspecifico() { return pesoEspecifico; }
} //Cierre del enum
```

```
/*
 * Esta clase prueba una clase de tipo Enum realizando distintas operaciones con ella test aprenderaprogramar.com
 * @author Alex Rodríguez
 * @version 13.04.19
 */
public class TestClaseEnum {
    public static void main (String[ ] Args) {
        TipoDeMadera maderaUsuario1 = TipoDeMadera.ROBLE;
        System.out.println ("La madera elegida por el usuario es " + maderaUsuario1.toString() + "\ncon un color " +
            maderaUsuario1.getColor() + " y con un peso específico de " + maderaUsuario1.getPesoEspecifico() + " kg/m3");
        System.out.println ("Un palet admite 2.27 m3 de volumen. A continuación el peso de los palets de las distintas maderas:");
        for (TipoDeMadera tmp: TipoDeMadera.values() ) {
            System.out.println (tmp.toString() + " el palet pesa " + (2.27f*(float)tmp.getPesoEspecifico() ) + " kg");
        }
    } //Cierre del main
} //Cierre de la clase
```

Escribe el código y ejecuta el método main. Obtenemos un resultado similar a este:

```
La madera elegida por el usuario es ROBLE
con un color Castaño verdoso y con un peso específico de 800 kg/m3
Un palet admite 2.27 m3 de volumen. A continuación el peso de los palets de las distintas maderas:
ROBLE el palet pesa 1816.0 kg      CAOBA el palet pesa 1747.9 kg
NOGAL el palet pesa 1861.4 kg      CEREZO el palet pesa 1793.2999 kg
BOJ el palet pesa 1532.25 kg
```

La posibilidad de disponer un constructor y métodos supone interesantes posibilidades para el trabajo con enumerados en Java.

EJERCICIO

Declara una clase que represente el tipo enumerado que represente tipos de piedra con los valores CALIZA, MARMOL, PIZARRA, CUARZITA. Si sus pesos específicos son de 1200, 1423.55, 1325 y 1466.22, crea un programa que muestre el peso de los palets de cada uno de los tipos de piedra. Puedes comprobar si tu código es correcto consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00683B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

MÉTODOS DE CLASE O STATIC FRENTE A MÉTODOS DE INSTANCIA.

Un método de instancia es el que se invoca **siempre** sobre una instancia (objeto) de una clase. Por ejemplo p1.getNombre(); siendo p1 un objeto de tipo Persona es un método de instancia: para invocarlo necesitamos una instancia de persona. **Un método de clase es aquel que puede ser invocado sin existir una instancia.**



Un método de clase se define agregando la palabra clave *static* antes del tipo en la signatura del método. Ejemplos:

```
//Ejemplo aprenderaprogramar.com

public static String getNombre () { ... }

public static int getNumeroDeDiasDelMes () { ... }
```

Los métodos de clase pueden ser invocados con la notación de punto de estas dos maneras:

```
//Ejemplo aprenderaprogramar.com

NombreDeLaClase.nombreDelMétodo (parámetros si los hay);

NombreDelObjeto.nombreDelMétodo (parámetros si los hay);
```

Por ejemplo si la clase Enero tuviera un método estático *getNumeroDeDiasDelMes()* podríamos escribir: *int dias = Enero.getNumeroDeDiasDelMes();*. La diferencia con un uso de método “normal” es que aquí no invocamos a un objeto, sino a una clase y por ello decimos que un método estático es un método de clase. Por ejemplo, los métodos de la clase Arrays del API de Java son métodos estáticos: no los invocamos sobre un objeto, sino sobre una clase. Otra clase que se basa en métodos estáticos es Math.

Por ejemplo el método `pow (double a, double b)` es un método que devuelve la potencia a^b . Es un método estático porque no se invoca sobre un objeto. Nosotros podemos crear métodos estáticos en nuestro código. Los interpretaremos como paquetes de código asociados a la clase.

BlueJ es un entorno de desarrollo especial que permite trabajar con objetos y métodos de instancia directamente. En otros entornos esto no es posible y debe indicársele al IDE una clase “de arranque”. Una vez se especifica esa clase, Java busca e invoca automáticamente el método main ubicado en dicha clase, y a partir de ese método comienza la creación de objetos y desarrollo de la aplicación. Los métodos de clase (estáticos) tienen estas limitaciones:

- a) No pueden acceder a campos de instancia (lógico, pues los campos van asociados a objetos).
- b) No pueden invocar a un método de instancia de la misma clase (lógico pues los métodos de instancia van asociados a objetos).

Ahora estamos en condiciones de reflexionar con un poco más de detenimiento sobre el método main. Recordemos su sintaxis: `public static void main (String[] Args)`. El método main será siempre un método público, ya que por su papel de iniciador de la aplicación no tendría sentido que fuera privado y por tanto inaccesible desde el exterior. **El método main siempre es un método estático, ya que cuando se invoca no existen objetos** creados con anterioridad porque la ejecución del programa aún no ha comenzado. Si un programa no ha comenzado, existen clases pero no objetos (aunque BlueJ es un entorno educativo especial que sí permite crear objetos e interactuar con ellos antes de ejecutar el método main, esto debemos verlo como algo no habitual en el desarrollo de programas). El tipo de `main` siempre será `void` (nulo) ya que no es un tipo función que devuelva un valor: su misión es arrancar la ejecución, no devolver un valor.

En el método main se incluye como parámetro para su invocación un array de Strings. Este array permitiría iniciar el programa con argumentos adicionales: por ejemplo podríamos indicarle una gama de colores de presentación entre varias posibles, o si se trata de un juego, si se ejecuta en modo 1 ó 2 jugadores, etc. En este caso suponiendo que la clase que contiene el main se llama *juego*, para la ejecución por consola escribiríamos > `java juego TwoPlayers Red`, donde *TwoPlayers* y *Red* son parámetros que condicionan la ejecución del programa. Cada palabra después del nombre de la clase se introduce en un array que se pasa al array `Args[]` que va como parámetro de la clase main. No obstante, es muy frecuente que los programas se inicien sin parámetros, para lo cual simplemente en consola habríamos de escribir *java* y el nombre de la clase.

En teoría, el cuerpo de un método main puede contener todo lo que se quiera y ser tan largo como se quiera. No obstante, ya hemos indicado que un buen diseño pasa por hacer el método main lo más corto posible y evitar que contenga la lógica del programa. En esencia, debe limitarse a crear objetos e invocar sus métodos. El código que controla la lógica del programa no tiene por qué estar en la clase con el método main. En main nos limitaríamos a crear un objeto que llevara el control del programa.

EJERCICIO

Intenta compilar el siguiente código:

```
public class Test {  
    int atributo1;  
    Test (int atrib) {atributo1 = 0;}  
    public static void main (String[ ] Args) {  
        System.out.println ("Mensaje 1");  
        System.out.println ("Atributo 1 vale" + this.getAtrib1());  
    } //Cierre del main  
    public int getAtrib1() {return atributo1;}  
} //Cierre de la clase
```

¿Qué mensaje de error se obtiene? ¿Por qué se obtiene este mensaje de error? ¿Cómo podemos solucionarlo para que se ejecute lo que pretendemos? Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com

Próxima entrega: CU00684B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

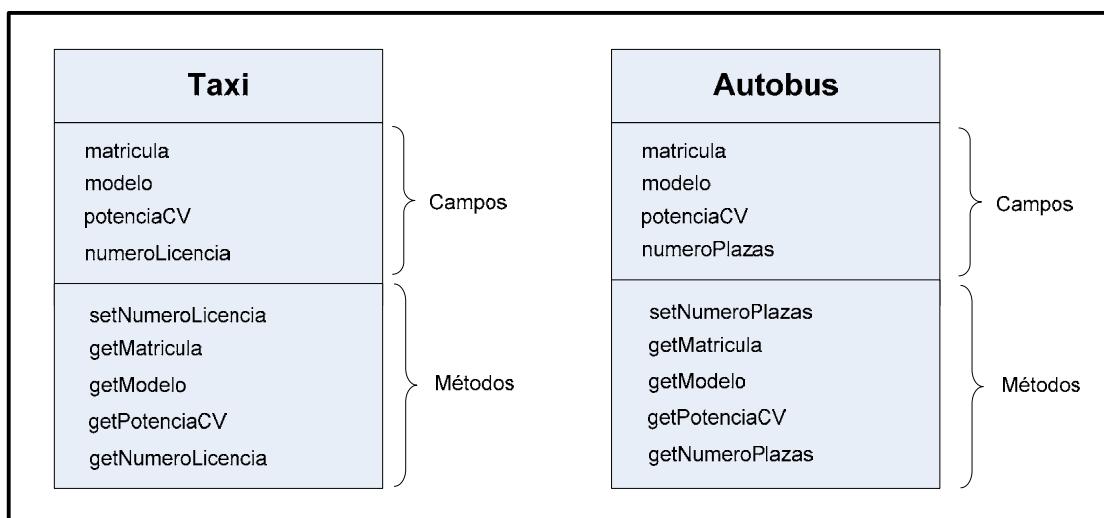
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

¿QUÉ ES LA HERENCIA EN PROGRAMACIÓN ORIENTADA A OBJETOS?

Muchas veces distintos objetos comparten campos y métodos que hacen aproximadamente lo mismo (por ejemplo almacenar y devolver un nombre del ámbito humano con el que se designa al objeto, como el título de un álbum de música, el título de un libro, el título de una película, etc.).



Por ejemplo en un proyecto que utilice objetos Taxi y objetos Autobus podríamos encontrarnos algo así:



Para una aplicación de gestión de una empresa de transporte que tenga entre sus vehículos taxis y autobuses podríamos tener otra clase denominada FlotaCirculante donde tendríamos posibilidad de almacenar ambos tipos de objeto (por ejemplo taxis en un ArrayList y autobuses en otro ArrayList) como reflejo de los vehículos que se encuentran en circulación en una fecha dada. Esas listas conllevarían una gestión para añadir o eliminar vehículos de la flota circulante, modificar datos, etc. resultando que cada una de las listas necesitaría un tratamiento o mantenimiento.

Si nos fijamos en el planteamiento del problema, encontramos lo siguiente:

- La definición de clases nos permite identificar **campos y métodos que son comunes** a Taxis y Autobuses. Si implementamos ambas clases tal y como lo venimos haciendo, incurriremos en duplicidad de código. Por ejemplo si el campo *matricula* es en ambas clases un tipo String, el código para gestionar este campo será idéntico en ambas clases.

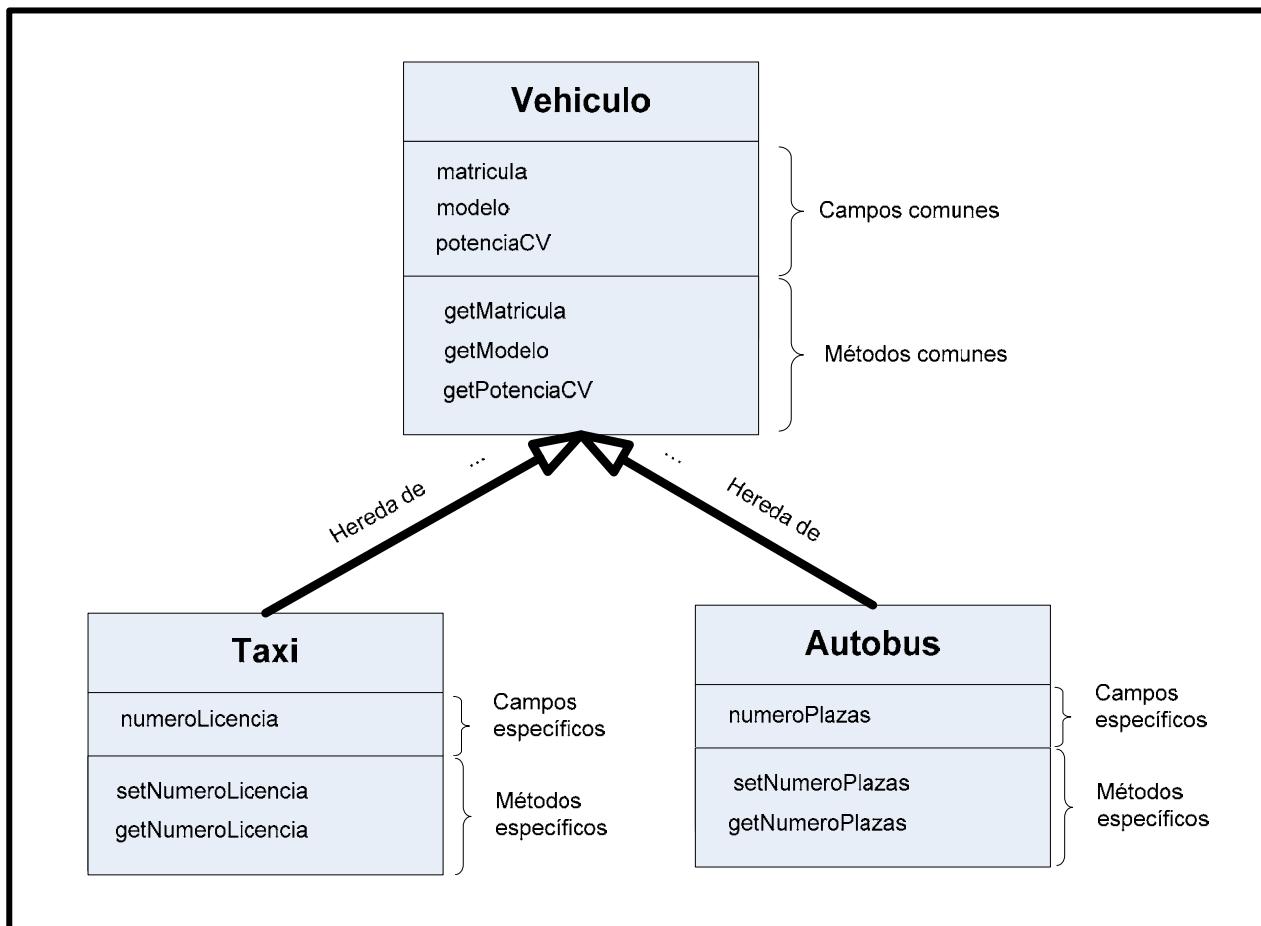
- b) La definición de clases nos permite identificar **campos y métodos que difieren** entre una clase y otra. Por ejemplo en la clase Taxi se gestiona información sobre un campo denominado *numeroDeLicencia* que no existe en la clase Autobus.
- c) Conceptualmente **podemos imaginar una abstracción** que engloba a Taxis y Autobuses: ambos podríamos englobarlos bajo la denominación de “Vehículos”. Un Taxi sería un tipo de Vehículo y un Autobus otro tipo de Vehículo.
- d) Si la empresa añade otros vehículos como minibuses, tranvías, etc. manteniendo la definición de clases tal y como la veníamos viendo, seguiríamos engrosando la **duplicidad de código**. Por ejemplo, un minibús también tendría matrícula, potencia... y los métodos asociados.

La duplicidad de código nos implicará problemas de mantenimiento. Por ejemplo inicialmente tenemos una potencia en caballos y posteriormente queremos definirla en kilowatios. O tenemos simplemente que modificar el código de un método que aparece en distintas clases. El tener el código duplicado nos obliga a tener que hacer dos o más modificaciones en sitios distintos. Pueden ser dos modificaciones, tres, cuatro o n modificaciones dependiendo del número de clases que se vieran afectadas, y esto a la larga genera errores al no ser el mantenimiento razonable.

En la clase FlotaCircular tambiéndremos seguramente duplicidades: por un lado un ArrayList de taxis y por otro un ArrayList de autobuses, por un lado una operación de adición de taxis y otra operación de adición de autobuses, por un lado una operación para mostrar los elementos de la lista de taxis y otra para los elementos de la lista de autobuses...

¿No sería más razonable, si una propiedad o método va a ser siempre común para varios tipos de objetos, que estuviera localizada en un sitio único del que ambos tipos de objeto “bebieran”? En los lenguajes con orientación a objetos la solución a esta problemática se llama herencia. La herencia es precisamente uno de los puntos clave de este tipo de lenguajes.

La herencia nos permite definir una clase como extensión de otra: de esta manera decimos “la clase 1.1 tiene todas las características de la clase 1 y además sus características particulares”. Todo lo que es común a ambas clases queda comprendido en la clase “superior”, mientras lo que es específico, queda restringido a las clases “inferiores”. En nuestro ejemplo definiríamos una clase denominada Vehiculo, de forma que la clase Taxi tuviera todas las propiedades de la clase Vehiculo, más algunas propiedades y métodos específicos. Lo mismo ocurriría con la clase Autobus y otras que pudieran “heredar” de Vehiculo. Podríamos seguir creando clases con herencia en un número indefinido: tantas como queramos. Si piensas en el API de Java, hay cientos de clases que heredan de clases jerárquicamente superiores como la clase Object. En un proyecto propio, podremos tener varias clases que hereden de una clase común.



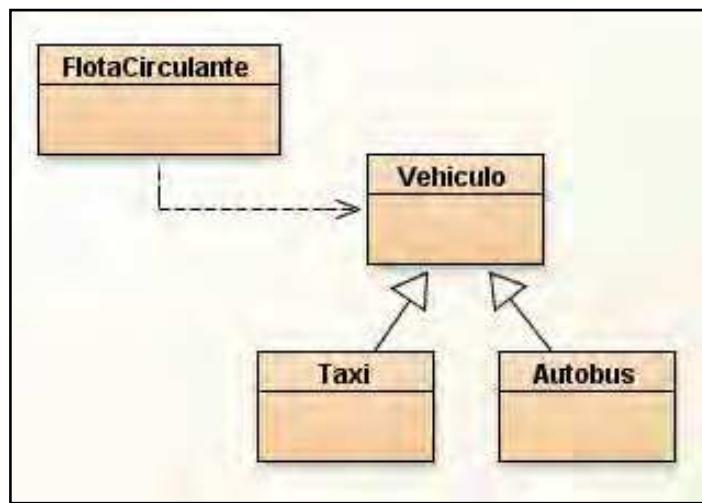
Esquema básico de herencia

Otro ejemplo: para la gestión de un centro educativo, podemos definir la clase Persona que comprende los campos y métodos comunes a todas las personas. Luego podremos definir la clase Estudiante, que es extensión de Persona, y comprende los campos y métodos de la clase superior, más algunos específicos. También podrían heredar de Persona la clase Profesor, la clase Director, la clase JefeDeEstudios, la clase PersonalAdministrativo, etc.

La primera aproximación a la herencia en Java la plantearemos para plasmar todo lo que hemos discutido en párrafos anteriores: en vez de definir cada clase por separado con operaciones o campos duplicados en cierta medida, definiremos una clase “padre” que contendrá todas las cosas que tengan en común otras clases. Luego definiremos las otras clases “hijo” como extensión de la clase padre, especificando para ellas únicamente aquello que tienen específico y distinto de la clase padre. Una característica esencial de la herencia es que permite evitar la **duplicidad de código**: aquello que es común a varias clases se escribe solo una vez (en la clase padre). La duplicidad de información, ya sea código o datos, es algo que por todos los medios hay que tratar de evitar en los sistemas informáticos por ser fuente de errores y de problemas de mantenimiento.

Si nos remitimos al esquema básico de herencia donde hemos representado las clases Vehículo, Taxi y Autobús, la clase Vehículo diremos que actúa como clase padre de las clases Taxi y Autobús. Vehículo recogería todo aquello que tienen o hacen en común taxis y autobuses. Aunque aquello que tienen en común se agrupa, tanto Taxi como Autobús tienen sus campos o métodos específicos.

Que una clase derive de otra en Java se indica mediante la palabra clave “**extends**”. Por eso muchas veces se usa la expresión “esta clase es extensión de aquella otra”. En los diagramas de clase la herencia se representa con una flecha de punta vacía. El aspecto en BlueJ sería algo así:



Este diagrama refleja que la clase **FlotaCirculante** usa a la clase **Vehiculo**, mientras que las clases **Taxi** y **Autobus** heredan de la clase **Vehiculo** (son extensión de la clase **Vehiculo**). Podemos decir que la clase hijo extiende (hace más extensa) a la clase padre. Una clase de la que derivan otras se denomina clase padre, clase base o superclase. Las clases que heredan se denominan clases derivadas, clases hijos o subclases. A partir de ahora los términos **superclase** y **subclase** son los que usaremos con más frecuencia. Una subclase podrá tener un acceso “más o menos directo” a los campos y métodos de la superclase en función de lo que defina el programador, como veremos más adelante. Para referirnos a la herencia también se usa la terminología “es un”. En concreto decimos que un objeto de la subclase es un objeto de la superclase, o más bien en casos concretos, que un **Taxi** es un **Vehiculo**, o que un **Estudiante** es una **Persona**. Sin embargo, al revés esto no es cierto, es decir, un **Vehiculo** no es un **Taxi**, ni una **Persona** es un **Estudiante**.

Dos subclases que heredan de una clase no deben tener información duplicada. Esto se consideraría un mal diseño. La información común debe estar en una superclase.

EJERCICIO

Se plantea desarrollar un programa Java que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos: productos frescos, productos refrigerados y productos congelados. Todos los productos llevan alguna información común como fecha de caducidad y número de lote, pero a su vez cada tipo de producto lleva alguna información específica, por ejemplo los productos congelados deben llevar la temperatura de congelación recomendada. Hay tres tipos de productos congelados: congelados por aire, congelados por agua y congelados por nitrógeno.

La empresa gestiona envíos a través de diferentes medios, y un envío puede contener cierto número de productos frescos, refrigerados o congelados. Identificar las 7 clases Java principales que podemos identificar dada la forma de funcionamiento de la empresa. Crear un esquema con las relaciones de herencia y/o uso entre las distintas clases.

Puedes comprobar si tu respuesta es correcta consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00685B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

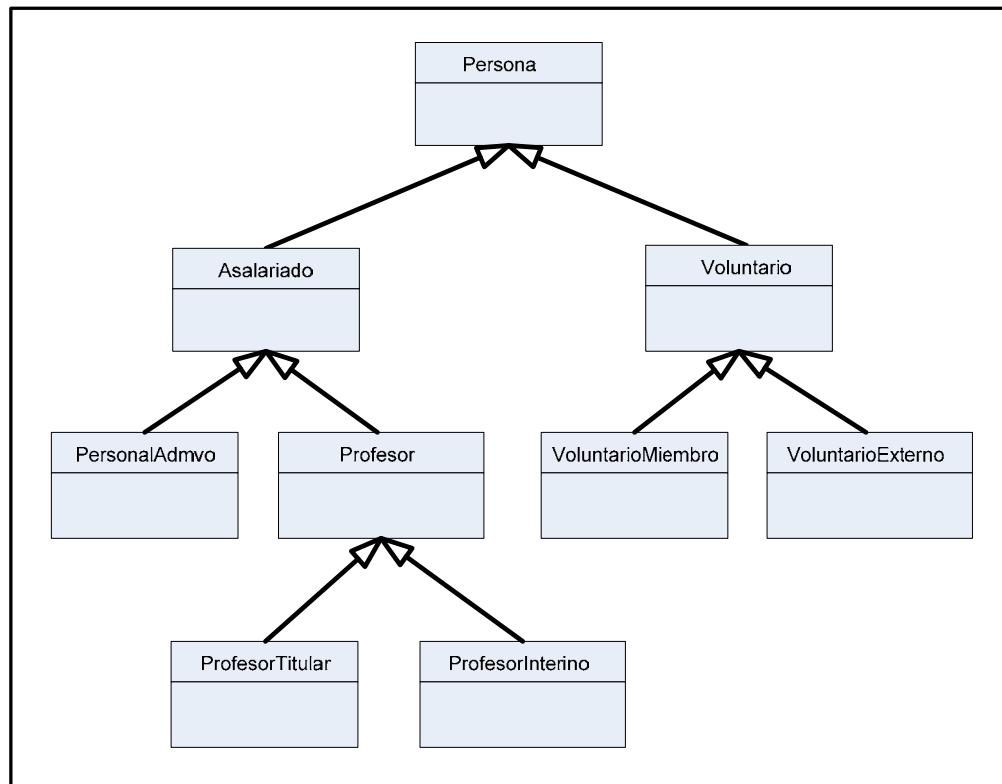
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

JERARQUÍAS DE HERENCIA EN JAVA. SUPERCLASES Y SUBCLASES.

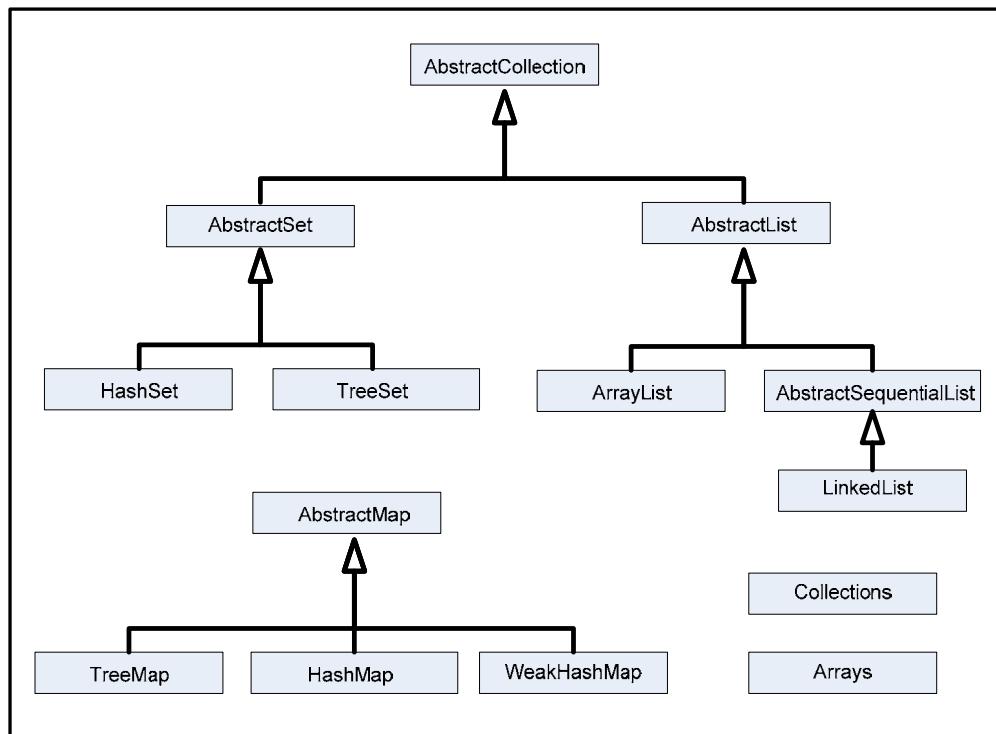
En Java muchas subclases pueden heredar de una misma superclase, y a su vez una subclase puede convertirse en superclase de otra. Así las cosas, podemos hablar de una jerarquía de herencia. La jerarquía es el esquema organizativo de las clases con relación de herencia entre sí.



En un proyecto nosotros definimos nuestra propia jerarquía de herencia.



Las clases del API de Java igualmente tienen establecida una jerarquía de herencia, en este caso definida por el equipo de desarrollo del lenguaje de la multinacional Oracle.



La herencia es una técnica de abstracción que nos permite agrupar objetos con características comunes. Todas las aplicaciones Java usan herencia, aunque no lo hagan de forma explícita. Esto es así porque cuando creamos una clase sin ninguna referencia a una superclase, el propio compilador inserta en la clase creada una relación de herencia directa con la clase `java.lang.Object`. Por este motivo todos los objetos, sean de la clase que sean, pueden invocar métodos de la clase `java.lang.Object` como `equals()` y `toString()`, aunque no siempre se van a obtener buenos resultados así.

Si quisieramos podríamos escribir para todas las clases `public class NombreDeLaClase extends Object`, aunque como es algo implícito a Java normalmente no lo escribiremos por ser redundante. En los diagramas representativos de la jerarquía de herencia ocurre lo mismo: Object siempre está en la cabecera del diagrama, aunque normalmente no se represente ya que se da por sobreentendido. Recordar que en Java los tipos primitivos no son objetos: no son instancias de clase, y por tanto no heredan de la superclase Object.

Los campos privados de una superclase no son accesibles por la subclase directamente. Decimos que la subclase no tiene derechos de acceso sobre los campos privados de la superclase. Para acceder a ellos tendrá que hacer uso de métodos de acceso o modificación. Una subclase puede invocar a cualquier método público de su superclase como si fuese propio. Lo veremos con ejemplos.

Próxima entrega: CU00686B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EJEMPLO DE HERENCIA EN JAVA. EXTENDS Y SUPER.

Para declarar la herencia en Java usamos la palabra clave *extends*. Ejemplo: *public class MiClase2 extends MiClase1*. Para familiarizarte con la herencia te proponemos que escribas y estudies un pequeño programa donde se hace uso de ella. Escribe el código de las clases que mostramos a continuación.



```
//Código de la clase Persona ejemplo aprenderaprogramar.com
public class Persona {
    private String nombre;
    private String apellidos;
    private int edad;

    //Constructor
    public Persona (String nombre, String apellidos, int edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }

    //Métodos
    public String getNombre () { return nombre; }
    public String getApellidos () { return apellidos; }
    public int getEdad () { return edad; }
} //Cierre de la clase
```

```
//Código de la clase profesor, subclase de la clase Persona ejemplo aprenderaprogramar.com
public class Profesor extends Persona {
    //Campos específicos de la subclase.
    private String IdProfesor;
    //Constructor de la subclase: incluimos como parámetros al menos los del constructor de la superclase
    public Profesor (String nombre, String apellidos, int edad) {
        super(nombre, apellidos, edad);
        IdProfesor = "Unknown"; } //Cierre del constructor

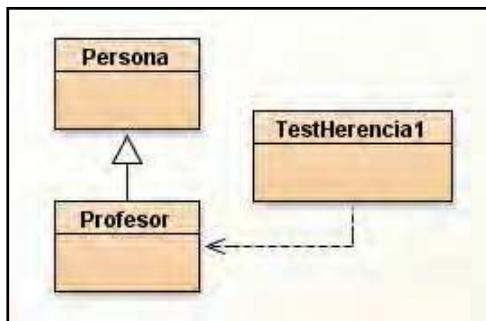
    //Métodos específicos de la subclase
    public void setIdProfesor (String IdProfesor) { this.IdProfesor = IdProfesor; }

    public String getIdProfesor () { return IdProfesor; }

    public void mostrarNombreApellidosYCarnet() {
        // nombre = "Paco"; Si tratáramos de acceder directamente a un campo privado de la superclase, salta un error
        // Sí podemos acceder a variables de instancia a través de los métodos de acceso públicos de la superclase
        System.out.println ("Profesor de nombre: " + getNombre() + " " + getApellidos() +
            " con Id de profesor: " + getIdProfesor() );
    } //Cierre de la clase
```

```
//Código de test aprenderaprogramar.com
public class TestHerencia1 {
    public static void main (String [ ] Args) {
        Profesor profesor1 = new Profesor ("Juan", "Hernández García", 33);
        profesor1.setIdProfesor("Prof 22-387-11");
        profesor1.mostrarNombreApellidosYCarnet();
    } //Cierre de la clase
```

El diagrama de clases y el resultado del test son del tipo que mostramos a continuación:



Profesor de nombre: Juan Hernández García
con Id de profesor: Prof 22-387-11

Los aspectos a destacar del código son:

- La clase persona es una clase “normal” definida tal y como lo venimos haciendo habitualmente mientras que la clase Profesor es una subclase de Persona con ciertas peculiaridades.
- Los objetos de la subclase van a tener campos *nombre*, *apellidos* y *edad* (heredados de Persona) y un campo específico *IdProfesor*. El constructor de una subclase ha de llevar obligatoriamente como parámetros al menos los mismos parámetros que el constructor de la superclase.
- El constructor de la subclase invoca al constructor de la superclase.** Para ello se incluye, obligatoriamente, la palabra clave super como primera línea del constructor de la subclase. La palabra super irá seguida de paréntesis dentro de los cuales pondremos los parámetros que requiera el constructor de la superclase al que queramos invocar. En este caso solo teníamos un constructor de superclase que requería tres parámetros. Si p.ej. hubiéramos tenido otro constructor que no requiriera ningún parámetro podríamos haber usado uno u otro, es decir, super(nombre, apellidos, edad) ó super(), o bien ambos teniendo dos constructores para la superclase y dos constructores para la subclase. Ejemplo:

En la superclase:

```
public Persona() {
    nombre = "";
    apellidos = "";
    edad = 0; }
```

```
public Persona (String nombre, String apellidos, int edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad; }
```

En la subclase:

```
public Profesor () {
    super();
    IdProfesor = "Unknown";}

public Profesor (String nombre, String apellidos, int edad) {
    super(nombre, apellidos, edad);
    IdProfesor = "Unknown"; }
```

Modifica el código de las clases Persona y Profesor para que queden con dos constructores tal y como hemos mostrado aquí. Crea objetos de ambos tipos en BlueJ y prueba sus métodos.

¿Qué ocurre si olvidamos poner super como primera línea de la subclase? Hay dos posibilidades: si la superclase tiene un constructor sin parámetros, el compilador incluirá en segundo plano super de forma automática y no saltará un error. De cualquier manera se considera contrario al buen estilo de programación, ya que no queda claro si se trata de un olvido. Por ello incluiremos siempre la palabra clave super. La otra posibilidad es que no haya un constructor sin parámetros, en cuyo caso saltará un error.

A modo de resumen: la inicialización de un objeto de una subclase comprende dos pasos. La invocación al constructor de la superclase (primera línea del constructor: super...) y el resto de instrucciones propias del constructor de la subclase.

EJERCICIO

Se plantea desarrollar un programa Java que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos: productos frescos, productos refrigerados y productos congelados. Todos los productos llevan esta información común: fecha de caducidad y número de lote. A su vez, cada tipo de producto lleva alguna información específica. Los productos frescos deben llevar la fecha de envasado y el país de origen. Los productos refrigerados deben llevar el código del organismo de supervisión alimentaria. Los productos congelados deben llevar la temperatura de congelación recomendada. Crear el código de las clases Java implementando una relación de herencia desde la superclase Producto hasta las subclases ProductoFresco, ProductoRefrigerado y ProductoCongelado. Cada clase debe disponer de constructor y permitir establecer (set) y recuperar (get) el valor de sus atributos y tener un método que permita mostrar la información del objeto. Crear una clase testHerencia2 con el método main donde se cree un objeto de cada tipo y se muestren los datos de cada uno de los objetos creados.

Puedes comprobar si tu respuesta es correcta consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00687B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EJERCICIO RESUELTO DE HERENCIA SIMPLE EN JAVA

En apartados anteriores del curso ya hemos comenzado con el estudio de la herencia en programación orientada a objetos. La herencia puede comprender numerosas clases, es decir, una clase puede heredar de otra clase que a su vez herede de otra clase. Pensemos ahora que un Profesor hereda de Persona, y que a su vez pueda ser ProfesorInterino o ProfesorTitular.

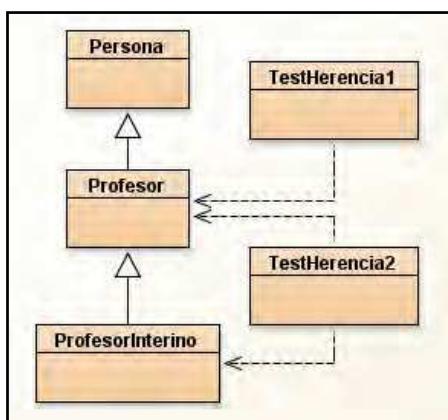


Trata de escribir el código de las clases ProfesorInterino y ProfesorTitular, de forma que hereden de profesor y que a su vez cada una tenga sus propios métodos. Una vez lo hayas hecho, escribe este código de ejemplo, la clase ProfesorInterino y una clase de test TestHerencia2 y estudia cómo hemos desarrollado la herencia.

```
//Código de la clase ProfesorInterino ejemplo aprenderaprogramar.com
import java.util.Calendar;
public class ProfesorInterino extends Profesor {
    private Calendar fechaComienzoInterinidad;
    public ProfesorInterino(Calendar fechainicioInterinidad) {
        super();
        fechaComienzoInterinidad = fechainicioInterinidad;
    }
    public ProfesorInterino(String nombre, String apellidos, int edad, Calendar fechainicioInterinidad) {
        super(nombre, apellidos, edad);
        fechaComienzoInterinidad = fechainicioInterinidad;
    }
    public Calendar getFechaComienzoInterinidad () { return fechaComienzoInterinidad;}
} //Cierre de la clase
```

```
import java.util.Calendar;
public class TestHerencia2 {
    public static void main (String [ ] Args) {
        Profesor profesor1 = new Profesor ("Juan", "Hernández García", 33);
        profesor1.setIdProfesor("Prof 22-387-11");
        profesor1.mostrarNombreApellidosYCarnet();
        Calendar fecha1 = Calendar.getInstance();
        fecha1.set(2019,10,22); //Los meses van de 0 a 11, luego 10 representa noviembre
        ProfesorInterino interino1 = new ProfesorInterino("José Luis", "Morales Pérez", 54, fecha1);
        System.out.println("El profesor interino 1 se incorporó en la fecha: " + fecha1.getTime().toString() );
    } //Cierre de la clase ejemplo aprenderaprogramar.com
```

El diagrama de clases y el resultado del test son:



Profesor de nombre: Juan Hernández García
con Id de profesor: Prof 22-387-11
El profesor interino 1 se incorporó en la fecha: Fri Nov 22 13:42:55 CET 2019

Los aspectos a destacar del código son:

- Hemos usado la clase Calendar del API de Java. La importamos, declaramos objetos de tipo Calendar y usamos algunos de sus métodos. No vamos a profundizar en esta clase porque no es ese nuestro objetivo: en este código es sólo una clase auxiliar para desarrollar este ejemplo.
- La clase ProfesorInterino tiene dos constructores y ambos invocan a la superclase Profesor mediante la palabra clave *super*. La superclase habrá de tener también dos constructores, ya que en caso contrario saltará un error.
- En el test creamos un ProfesorInterino pasando 4 parámetros al constructor: tres de esos parámetros son gestionados por el constructor de la superclase, y el restante es gestionado por el constructor de la subclase.

EJERCICIO

Se plantea desarrollar un programa Java que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos: productos frescos, productos refrigerados y productos congelados. Todos los productos llevan esta información común: fecha de caducidad y número de lote. A su vez, cada tipo de producto lleva alguna información específica. Los productos frescos deben llevar la fecha de envasado y el país de origen. Los productos refrigerados deben llevar el código del organismo de supervisión alimentaria, la fecha de envasado, la temperatura de mantenimiento recomendada y el país de origen. Los productos congelados deben llevar la fecha de envasado, el país de origen y la temperatura de mantenimiento recomendada.

Hay tres tipos de productos congelados: congelados por aire, congelados por agua y congelados por nitrógeno. Los productos congelados por aire deben llevar la información de la composición del aire con que fue congelado (% de nitrógeno, % de oxígeno, % de dióxido de carbono y % de vapor de agua). Los productos congelados por agua deben llevar la información de la salinidad del agua con que se realizó la congelación en gramos de sal por litro de agua. Los productos congelados por nitrógeno deben llevar la información del método de congelación empleado y del tiempo de exposición al nitrógeno expresada en segundos.

Crear el código de las clases Java implementando una relación de herencia siguiendo estas indicaciones:

- a) En primer lugar realizar un esquema con papel y bolígrafo donde se represente cómo se van a organizar las clases cuando escribamos el código. Estudiar los atributos de las clases y trasladar a la superclase todo atributo que pueda ser trasladado.
- b) Crear superclases intermedias (aunque no se correspondan con la descripción dada de la empresa) para agrupar atributos y métodos cuando sea posible. Esto corresponde a “realizar abstracciones” en el ámbito de la programación, que pueden o no corresponderse con el mundo real.
- c) Cada clase debe disponer de constructor y permitir establecer (set) y recuperar (get) el valor de sus atributos y tener un método que permita mostrar la información del objeto cuando sea procedente.

Crear una clase `testHerencia3` con el método `main` donde se creen: dos productos frescos, tres productos refrigerados y cinco productos congelados (2 de ellos congelados por agua, otros 2 por agua y 1 por nitrógeno). Mostrar la información de cada producto por pantalla.

Puedes comprobar si tu respuesta es correcta consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00688B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

TIPOS Y SUBTIPOS. POLIMORFISMO Y VARIABLES POLIMÓRFICAS

Al igual que se forma una jerarquía de clases, el hecho de que las clases definan tipos hace que la herencia dé lugar a una jerarquía de tipos. El tipo que se define mediante una subclase se dice que es **un subtipo del tipo definido en su superclase**.



Los supertipos pueden usarse para definir operaciones que admitan objetos de distintos subtipos. Por ejemplo, podemos crear una colección que admite objetos de distintos subtipos:

`ArrayList <Profesor> ListadoProfesores = new ArrayList <Profesor>();` supone que admitimos en el listado de profesores a todos los subtipos de Profesor, por tanto podemos incluir en el listado tanto a profesores interinos como a profesores titulares. No necesitamos dos listados diferentes para cada tipo de profesor. Podemos tener un listado único si así lo deseamos. Esto nos permite unificar las operaciones con listados de profesores en una sola clase sin tener que duplicar código para profesores titulares y para profesores interinos. Por ejemplo mostrar los componentes de la lista, añadir componentes a la lista, etc. van a ser operaciones comunes tanto para profesores titulares como interinos.

Una variable que apunta a un objeto de un supertipo puede contener objetos de ese supertipo (si es que es coherente que existan) o de cualquier subtipo en escalas dependientes dentro de la jerarquía de tipos. Así resultarían válidas declaraciones como estas (suponiendo que se admiten constructores sin parámetros):

```
Persona p1 = new Persona();
```

```
Persona p1 = new Profesor();
```

```
Persona p3 = new ProfesorInterino();
```

Al uso de variables de subtipos en lugares donde se espera (o se admite) un objeto de un supertipo se le denomina “sustitución”. Los lenguajes orientados a objetos trabajan con el principio de sustitución: los tipos hijos pueden sustituir a los tipos padres. Sin embargo, la operación contraria no es posible: un tipo padre no puede ocupar el lugar de un tipo hijo.

Esto sería erróneo: `ProfesorInterino p1 = new Persona();`. La persona puede ser un profesor interino o no: existe la incertidumbre de que lo sea o no lo sea. Java no puede saber si la persona es profesor interino o no, por lo que diremos que esta asignación no es válida en Java. También sería erróneo declarar `ProfesorInterino p1 = new ProfesorTitular();`. Obviamente esto no tiene sentido.

Para el API de Java, también es aplicable lo indicado: `List <String> miListado = new ArrayList <String>();` es válido porque `ArrayList` es un subtipo de `List`. En ocasiones declararemos un tipo `List` sin tener predefinido si lo vamos a implementar en un subtipo `ArrayList` o en otro subtipo como `LinkedList`. A su vez, un método que espere un objeto de tipo `List` admitirá recibir tanto un `ArrayList` como un `LinkedList`.

Java decimos que se basa en el **polimorfismo**, entre otras razones porque una variable que apunta a un objeto admite distintas formas de ese objeto: las formas definidas por la superclase y las clases que extienden a la superclase. Es decir, una variable que apunta a un objeto es polimórfica porque admite distintos tipos de objetos, no solo uno. Una variable de tipo `Persona` admite tanto objetos de tipo `Persona`, como objetos de tipo `Profesor`, como objetos de tipo `ProfesorInterino`. O visto de otra manera, podríamos decir que un `ProfesorInterino` es una forma de `Profesor`, pero también una forma de `Persona`, y también una forma de `Object`. Las variables pueden ser polimórficas. El polimorfismo también se concreta de otras maneras en Java, como veremos más adelante.

La posibilidad de uso de variables polimórficas reduce la cantidad de código que es necesario escribir y facilita la reusabilidad. Por ejemplo no necesitamos dos bucles:

```
for (ProfesorInterino tmp : listado) { ... }
```

```
for (ProfesorTitular tmp : listado) { ... }
```

Sino solo uno:

```
for (Profesor tmp: listado) { ... }
```

La variable `listado`, que apuntará a un objeto tipo `ArrayList` o similar, puede contener profesores interinos o profesores titulares, pero gracias al polimorfismo el bucle nos funciona en ambos casos, porque admite tanto profesores como cualquiera de los subtipos de profesores.

EJERCICIO

Amplía el código del programa Java que planteamos como ejercicio en la entrega CU00687 de este curso, relativo a la gestión de una empresa agroalimentaria, teniendo en cuenta que la empresa gestiona envíos a través de diferentes medios, y un envío puede contener cierto número de productos frescos, refrigerados o congelados. Añade al código:

- a) Una clase EnvioDeProductos que represente un envío de productos como colección de objetos que admite el polimorfismo.
- b) Crear una clase testHerencia4 con el método main donde se creen: dos productos frescos, tres productos refrigerados y cinco productos congelados (2 de ellos congelados por agua, otros 2 por agua y 1 por nitrógeno). Crear un envío que represente la agrupación de los anteriores productos. Mostrar por pantalla la información del número de productos que componen el envío y recorrer los productos del envío usando iterator para mostrar la información (valor de los atributos) de cada uno de ellos.

Puedes comprobar si tu respuesta es correcta consultando en los foros aprenderaprogramar.com.

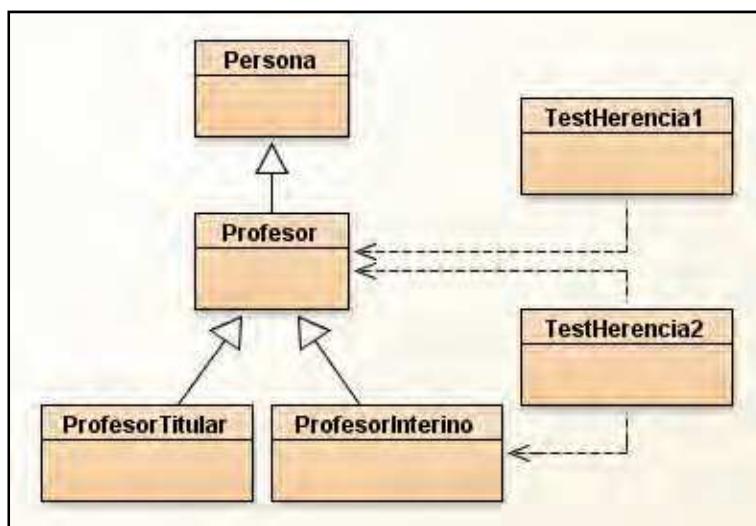
Próxima entrega: CU00689B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONVERSIÓN DE TIPOS. CASTING. CLASSCASTEXCEPTIONS.

Java admite la conversión de tipos con ciertas limitaciones. Consideremos una jerarquía de herencia como la que vemos en el siguiente diagrama de clases, sobre el que vamos a analizar las posibilidades de conversión de tipos de distintas formas.



1. Conversión hacia arriba

Se trataría por ejemplo de poner lo que está a un nivel inferior en un nivel superior, por ejemplo poner un profesor interino como profesor. Posible código: `profesor43 = profesorinterino67;`

Ponemos lo que está abajo (el profesor interino) arriba (como profesor). Esto es posible, pero dado que lo que está abajo generalmente contiene más campos y métodos que lo que está arriba, perderemos parte de la información. Sobre el objeto `profesor43` ya no podremos invocar los métodos propios de los profesores interinos.

2. Conversión hacia abajo

Se trataría de poner lo que está arriba abajo, por ejemplo poner un Profesor como ProfesorInterino. Esto no siempre es posible. El supertipo admite cualquier forma (es polimórfico) de los subtipos: si el supertipo almacena el subtipo al que queremos realizar la conversión, será posible usando lo que se

denomina “Enmascaramiento de tipos” o “Hacer Casting” (cast significa “moldear”). Si el supertipo no almacena el subtipo al que queremos convertirlo, la operación no es posible y saltará un error. Ejemplo:

Profesor p1; //p1 es tipo Profesor. Admite ser Profesor, ProfesorTitular o ProfesorInterino.

ProfesorInterino p44 = new ProfesorInterino(); //p44 es ProfesorInterino.

p1 = p44; // Conversión hacia arriba: sin problema. Ahora p1 que es tipo profesor, almacena un profesor interino

p44 = p1 // ERROR en la conversión hacia abajo. El compilador no llega tan lejos como para saber si p1 almacena un profesor interino u otro tipo de profesor y ante la incertidumbre salta un error. La forma de forzar al compilador a que “comprenda” que p1 está almacenando un profesor interino y por tanto puede asignarse a una variable que apunta a un profesor interino se llama “hacer casting”. Escribiríamos lo siguiente: *p44 = (ProfesorInterino) p1;*

El nombre de un tipo entre paréntesis se llama “operador de enmascaramiento de tipos” y a la operación en sí la llamamos enmascaramiento o hacer casting. El casting es posible si el objeto padre contiene a un objeto hijo, **pero si no es así aunque la compilación sea correcta en tiempo de ejecución saltará un error “ClassCastException”**, o error al hacer cast con las clases. No siempre es fácil determinar a qué tipo de objeto apunta una variable. Por ello el casting o enmascaramiento debiera evitarse en la medida de lo posible ya que el que un programa compile pero contenga potenciales errores en tiempo de ejecución es algo no deseable. Un programa bien estructurado normalmente no requerirá hacer casting reiteradamente, o lo hará de forma muy controlada. Si durante la escritura de un programa nos vemos en la necesidad de realizar casting, debemos plantearnos la posibilidad de reestructurar código para evitarlo.

Cuando incluyamos conversiones de tipos usando casting en nuestro código, para evitar errores conviene filtrar las operaciones de enmascaramiento asegurándonos antes de hacerlas de que el objeto padre contiene un hijo del subtipo al que queremos hacer la conversión. Esta verificación se hace usando la palabra clave *instanceof* como explicaremos a continuación.

3. Conversión de lado a lado

Se trataría de poner lo que está a un lado al otro lado, por ejemplo convertir un ProfesorInterino en ProfesorTitular o al revés. Esto no es posible en ningún caso.

DETERMINACIÓN DEL TIPO DE VARIABLES CON INSTANCE OF

La palabra clave *instanceof*, todo en minúsculas, sirve para verificar el tipo de una variable. La sintaxis que emplearemos para *instanceof* y sus normas de uso serán las siguientes:

```
if (profesor43 instanceof ProfesorInterino) {...} else { ...}
```

a) Sólo se pueden comparar instancias que relacionen dentro de la jerarquía de tipos (en cualquier dirección) pero no objetos que no relacionen en una jerarquía. Es decir, no podemos comparar profesores con taxis por ejemplo, porque no relacionarán dentro de una jerarquía.

b) Solo se puede usar instanceof asociado a un condicional. No se puede usar por ejemplo directamente en una impresión por pantalla con System.out.println(...).

Ejemplos de sintaxis y ejemplo de código. Escribe y compila el código y comprueba el resultado:

```
if (profesor73 instanceof ProfesorInterino) --> la sintaxis es válida.  
if (interino1 instanceof ProfesorInterino) --> la sintaxis es válida.  
if (interino1 instanceof Profesor) --> la sintaxis es válida.  
if (fecha1 instanceof Profesor) --> Error: las instancias no están en una jerarquía de tipos
```

```
import java.util.Calendar;  
//Test conversión de tipos. Ejemplo de código aprenderaprogramar.com  
public class TestHerencia3 {  
    public static void main (String [ ] Args) {  
        Profesor profesor1 = new Profesor ("Juan", "Hernández García", 33);  
        Calendar fecha1 = Calendar.getInstance();  
        fecha1.set(2019,10,22); //Los meses van de 0 a 11, luego 10 representa noviembre  
        ProfesorInterino interino1 = new ProfesorInterino("José Luis", "Morales Pérez", 54, fecha1);  
  
        Profesor profesor73 = interino1; //Ahora el supertipo contiene un subtipo, en principio con pérdida de información  
        if (profesor73 instanceof ProfesorInterino) {  
            System.out.println ("***profesor73 es un objeto de tipo ProfesorInterino" );}  
        if (profesor73 instanceof Profesor) { System.out.println ("profesor73 es un objeto de tipo Profesor ¡ES POLIMÓRFICO!" ); }  
  
        if (interino1 instanceof Profesor) { System.out.println ("interino1 es un objeto de tipo Profesor ¡ES POLIMÓRFICO TAMBIÉN!" );  
        } else { System.out.println ("interino1 no apunta a un objeto de tipo Profesor" ); }  
  
        if (profesor1 instanceof ProfesorInterino) {  
            System.out.println ("profesor1 es un objeto de tipo ProfesorInterino" );  
        } else { System.out.println ("profesor1 no es un objeto de tipo ProfesorInterino. Nunca ha sido un interino." ); }  
    } //Cierre del main  
} //Cierre de la clase ejemplo aprenderaprogramar.com
```

```
***profesor73 es un objeto de tipo ProfesorInterino  
profesor73 es un objeto de tipo Profesor ¡ES POLIMÓRFICO!  
interino1 es un objeto de tipo Profesor ¡ES POLIMÓRFICO TAMBIÉN!  
profesor1 no es un objeto de tipo ProfesorInterino. Nunca ha sido un interino.
```

Próxima entrega: CU00690B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

SOBREESCRIBIR MÉTODOS EN JAVA. MÉTODOS POLIMÓRFICOS.

En apartados anteriores del curso hemos visto conceptos como herencia en Java, jerarquías de herencia entre clases, polimorfismo y variables polimórficas. Vamos a centrarnos ahora en las variables. Ya sabemos que en Java las variables que representan un objeto no son el objeto mismo, sino referencias al objeto.



Cuando hablamos de una variable que apunta a un objeto podemos distinguir, debido a la herencia, dos tipos:

- a) El tipo declarado para la variable en el código fuente: se llama **tipo estático**.
- b) El tipo del objeto al que apunta la variable en un momento dado (en tiempo de ejecución): puede ser el tipo declarado o un subtipo del tipo declarado, y se llama **tipo dinámico**.

Ejemplo:

```
ProfesorInterino profesorInterino43 = new ProfesorInterino();  
Persona p1 = new Persona(); p1 = profesorInterino43;
```

El tipo estático de p1 es Persona, mientras que tras la ejecución de la tercera instrucción su tipo dinámico pasa a ser ProfesorInterino porque el objeto al que pasa a apuntar es a un profesor interino. Digamos que inicialmente el contenedor de p1 es una caja de tipo “Persona” y que en la tercera línea la caja pasa a ser de tipo “ProfesorInterino”.

Ejemplo:

```
Persona p1 = new ProfesorInterino();
```

El tipo estático es Persona y el tipo dinámico es ProfesorInterino. ¿Por qué? Porque estamos creando una variable de tipo Persona e inmediatamente asignándole de forma dinámica un objeto anónimo de tipo ProfesorInterino. Sin embargo, no podemos invocar un método exclusivo de ProfesorInterino sobre p1 porque el compilador se basa en los tipos estáticos para hacer comprobaciones.

El compilador controla los tipos basándose en el tipo estático (declarado): no conoce si en un momento dado una variable está apuntando a un subtipo. Debido a esto, no podremos llamar a un método de un subtipo si la variable corresponde a una superclase, aunque la variable apunte a la subclase, porque el compilador no llega a conocer a qué tipo apunta la variable en tiempo de ejecución. Como veremos a continuación, esto es una limitación relativa, porque en los métodos que

denominamos “sobreescritos” sí se llega a identificar si un método corresponde a un supertipo o a un subtipo.

Decimos que un método está sobreescrito cuando está presente, exactamente con la misma signatura, en una subclase y en una superclase. En esta situación un objeto de la subclase tiene dos métodos con el mismo nombre y la misma signatura: uno heredado de la superclase y otro definido en su propia estructura. Una subclase puede sobreescibir la implementación de un método. Para hacerlo, la subclase declara un método con la misma signatura que la superclase, pero con un cuerpo diferente (hacerlo con el mismo cuerpo sería repetitivo y no tendría mucho sentido). El método sobreescrito tiene precedencia cuando se invoca sobre objetos de la subclase. Por el contrario, para los objetos de la subclase el método de la superclase pierde visibilidad.

Si como hemos dicho, el compilador se basa en el tipo estático para su trabajo, podríamos pensar que si invocamos a un objeto de tipo estático “superclase” y tipo dinámico “subclase” con un método sobreescrito, el método que se utilice sería el propio de la superclase. Pero sin embargo, esto no es así: el control de tipos del compilador se basa en los tipos estáticos pero en tiempo de ejecución los métodos que se ejecutan dan preferencia al tipo dinámico. Es decir, **en tiempo de ejecución Java está constantemente “buscando” (“ligando” o “despachando”) el método que corresponda en función del tipo dinámico** al que apunta una variable. Si el método invocado no se encuentra definido en el tipo dinámico de que se trate, Java busca el método en la superclase para tratar de ejecutarlo. Si no lo encuentra en la superclase, continúa subiendo niveles de clase hasta alcanzar a la clase Object. Si en la clase Object tampoco se encontrara un método con ese nombre, el programa no llegaría a compilar. Decimos que los métodos en Java son polimórficos porque una misma llamada en un programa puede dar lugar a la ejecución de distintos métodos según el tipo dinámico de una variable. Ya tenemos una segunda forma de expresión del polimorfismo en Java: además de polimorfismo en variables, hablamos de polimorfismo en métodos.

Tenemos que distinguir dos momentos temporales en un programa, la compilación durante la cual el compilador realiza una serie de verificaciones y transforma el código fuente en código máquina, y la ejecución, durante la cual una variable puede cambiar de tipo debido al polimorfismo que admite Java. El compilador solo conoce un tipo: el tipo estático o declarado en el código fuente. No obstante, durante la ejecución la máquina virtual Java es capaz de identificar el tipo dinámico de las variables que apuntan a objetos y asignar un método u otro en función de ese tipo dinámico.

Esto tiene ciertos riesgos. Si pensamos que hemos sobreescrito un método para que haga una tarea específica por nosotros requerida, pero no es así o lo hemos borrado, aparentemente el programa no tendrá problemas de compilación si se encuentra el método en una superclase. Sin embargo, los resultados pueden no ser los deseados y esto generarnos un quebradero de cabeza al no saber determinar con precisión por qué ocurre lo que ocurre. Debido a ello, muchos programadores tienen por costumbre dentro de los primeros pasos al definir una clase, sobreescibir algunos métodos como *toString* y *equals* para esos métodos respondan en la clase exactamente como se pretende y no de acuerdo a una especificación de una superclase que puede no ser adecuada.

EJERCICIO

Responde a las siguientes preguntas:

Supongamos la superclase Vehiculo que representa a cualquier tipo de vehículo y la subclase Taxi que representa a un tipo de vehículo concreto.

- a) ¿Un objeto de tipo estático declarado Taxi puede contener a un objeto Vehiculo en tiempo de ejecución?
- b) ¿Un objeto de tipo estático declarado Vehiculo puede contener a un objeto Taxi en tiempo de ejecución?
- c) Escribe el código de una clase Vehiculo con los atributos matricula (String) y numeroDeRuedas (int), constructor, métodos getters y setters y método toString() para mostrar la información de un vehículo.
- d) Escribe el código de una clase Taxi que herede de Vehiculo y que además de los atributos de Vehiculo lleve un atributo adicional nombreDelConductor (String) y numeroDePlazas (int), constructor, métodos getters y setters y método toString() para mostrar la información de un Taxi.
- e) Escribe el código de una clase test con el método main que cree un objeto cuyo tipo es Vehiculo, instanciado como Taxi. Establece valores para sus atributos y usa el método toString(). ¿Qué método toString() resulta de aplicación, el propio de la clase Vehiculo o el propio de la clase Taxi? ¿Por qué?

Puedes comprobar si tus respuestas son correctas consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00691B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

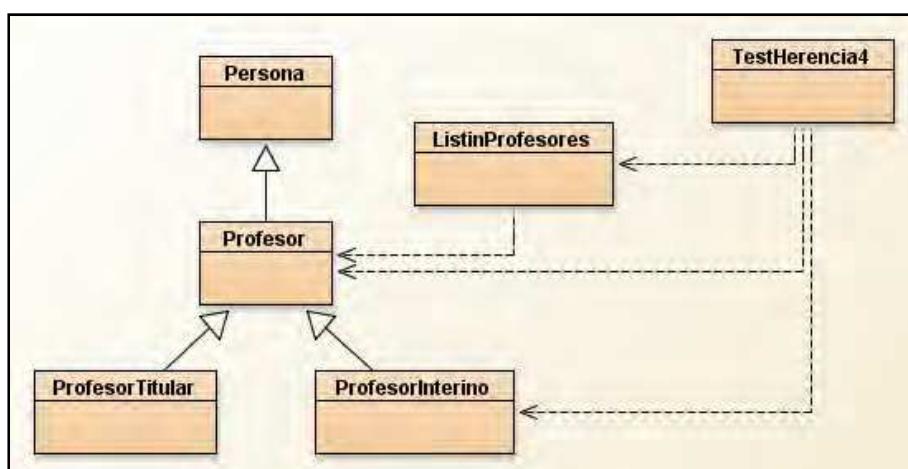
http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

EJERCICIO EJEMPLO RESUELTO POLIMORFISMO, HERENCIA, SOBREESCRITURA.

En apartados anteriores del tutorial hemos visto conceptos como herencia en Java, polimorfismo y sobreescritura de métodos. Vamos a plantear y desarrollar un ejercicio donde, a partir de un diagrama de clases, definimos el código que usa todos los conceptos estudiados.



Analiza el siguiente diagrama de clases. En él se pueden observar relaciones de herencia y relaciones de uso:



Trata de definir el código de las clases, estableciendo las relaciones de herencia y uso entre ellas. Trata de crear una clase con el método main (**TestHerencia4**) donde de alguna manera crees objetos de los distintos tipos y hagas uso de ellos, por ejemplo crea profesores interinos y titulares y luego recórrrelos con un for extendido donde el tipo sea Profesor (uso del polimorfismo). Luego compáralo con las explicaciones y soluciones que damos a continuación.

En la solución que hemos planteado nosotros, en el tipo Profesor hemos incluido un método denominado `mostrarDatos()` que muestra los datos propios de un objeto Profesor. Luego, en las subclases ProfesorInterino y ProfesorTitular hemos sobreescrito el método `mostrarDatos()` de modo que en este caso únicamente muestra los datos específicos de los subtipos.

Por último, en la clase **ListinProfesores** simulamos un listín que admite todo tipo de profesores mediante un `ArrayList` que usa objetos de tipo **Profesor**, y que permite listar los profesores mediante un método `listar()` que lo que hace es invocar el método `mostrarDatos()` de los objetos contenidos en la lista. Si el método utilizado se basara en el tipo declarado en el código, `listar()` siempre nos devolvería

los datos de los objetos Profesor. Sin embargo, como veremos, esto no es así: cuando la variable apunta a un subtipo, el método invocado en tiempo de ejecución es el propio del subtipo, mientras que cuando la variable apunta a un tipo sí se invoca el método propio del tipo. Por eso decimos que Java hace una búsqueda dinámica del método: el método que se usa depende del tipo dinámico del objeto. Escribe este código:

```
//Código de la clase Persona ejemplo aprenderaprogramar.com
public class Persona {
    private String nombre; private String apellidos; private int edad;

    public Persona() { nombre = ""; apellidos = ""; edad = 0; }
    public Persona (String nombre, String apellidos, int edad) {
        this.nombre = nombre; this.apellidos = apellidos; this.edad = edad; }
    public String getNombre() { return nombre; }
    public String getApellidos () { return apellidos; }
    public int getEdad() { return edad; }
} //Cierre de la clase
```

```
public class Profesor extends Persona { //Ejemplo aprenderaprogramar.com
    private String IdProfesor;

    public Profesor () { super();
        IdProfesor = "Unknown";}

    public Profesor (String nombre, String apellidos, int edad) {
        super(nombre, apellidos, edad);
        IdProfesor = "Unknown"; }

    public void setIdProfesor (String IdProfesor) { this.IdProfesor = IdProfesor; }
    public String getIdProfesor () { return IdProfesor; }
    public void mostrarDatos() {
        System.out.println ("Datos Profesor. Profesor de nombre: " + getNombre() + " " + getApellidos() +
        " con Id de profesor: " + getIdProfesor() );
    }
} //Cierre de la clase ejemplo aprenderaprogramar.com
```

```
import java.util.Calendar; //Ejemplo aprenderaprogramar.com
public class ProfesorInterino extends Profesor {

    private Calendar FechaComienzoInterinidad;
    public ProfesorInterino(Calendar fechaComienzoInterinidad) {
        super();
        FechaComienzoInterinidad = fechaComienzoInterinidad; }
    public ProfesorInterino (String nombre, String apellidos, int edad, Calendar fechaComienzoInterinidad) {
        super(nombre, apellidos, edad);
        FechaComienzoInterinidad = fechaComienzoInterinidad; }

    public Calendar getFechaComienzoInterinidad () { return FechaComienzoInterinidad; }
    public void mostrarDatos() { System.out.println("Datos ProfesorInterino. Comienzo interinidad: " +
    FechaComienzoInterinidad.getTime().toString() ); }
} //Cierre de la clase
```

```

import java.util.ArrayList; //Ejemplo aprenderaprogramar.com
public class ListinProfesores{
    private ArrayList <Profesor> listinProfesores;

    //Constructor
    public ListinProfesores () {
        listinProfesores = new ArrayList <Profesor> ();
    }

    //Métodos
    public void addProfesor (Profesor profesor) {
        listinProfesores.add(profesor); } // Cierre método addProfesor

    public void listar() {
        System.out.println ("Se procede a mostrar los datos de los profesores existentes en el listín");
        for (Profesor tmp: listinProfesores) { //Uso de for extendido
            tmp.mostrarDatos(); }
    } //Cierre método
} //Cierre de la clase

```

```

import java.util.Calendar; //Ejemplo aprenderaprogramar.com
public class TestHerencia4 {
    public static void main (String [ ] Args) {

        Profesor profesor1 = new Profesor ("Juan", "Hernández García", 33);
        profesor1.setIdProfesor("Prof 22-387-11");

        Calendar fecha1 = Calendar.getInstance();
        fecha1.set(2019,10,22); //Los meses van de 0 a 11, luego 10 representa noviembre
        ProfesorInterino interino1 = new ProfesorInterino("José Luis", "Morales Pérez", 54, fecha1);

        ListinProfesores listin1 = new ListinProfesores ();
        listin1.addProfesor(profesor1);
        listin1.addProfesor(interino1);
        listin1.listar(); } //Cierre del main

    } //Cierre de la clase

```

Se procede a mostrar los datos de los profesores existentes en el listín

Datos Profesor. Profesor de nombre: Juan Hernández García con Id de profesor: Prof 22-387-11

Datos ProfesorInterino. Comienzo interinidad: Fri Nov 22 12:28:44 CET 2019

No hemos incluido el código de ProfesorTitular porque no lo utilizamos en el test. Si observamos el código de la clase ProfesorInterino vemos que el método mostrarDatos() está sobreescrito respecto al de su superclase Profesor. El método mostrarDatos() de la clase Profesor muestra nombre, apellidos e id de profesor, mientras que el método mostrarDatos() de la clase ProfesorInterino muestra la fecha de comienzo de la interinidad. En la clase ListinProfesores definimos un tipo que almacena objetos de tipo Profesor y su método listar() invoca el método mostrarDatos(). En la clase TestHerencia4 creamos un objeto Profesor y un objeto ProfesorInterino e introducimos ambos en un objeto de tipo ListinProfesores. ¿Qué ocurre cuando invocamos a listar() en el objeto donde tenemos una colección

con objetos de la superclase (Profesores) y objetos de la subclase (ProfesoresInterinos)? Que según sea el tipo dinámico del objeto, se usa el método mostrarDatos() con mayor cercanía. En el caso del objeto ProfesorInterino, se usa el método propio de los profesores interinos, aunque hayamos dicho que el ArrayList contiene objetos Profesor y aunque el bucle que invoca al método mostrarDatos() indique que el tipo que se usa es Profesor. Esto es debido a que toda variable de tipo Profesor es polimórfica y admite objetos de distintos tipos. A la hora de diseñar y dar nombres a las clases debes usar la lógica de la herencia y evitar nombres o relaciones entre clases que resulten contradictorios o contrarios a lo que sería lógico en el mundo real.

`tmp.mostrarDatos()` puede dar lugar a la ejecución del método de la clase ProfesorInterino, ProfesorTitular o Profesor, dependiendo del tipo dinámico al que apunte la variable.

Los objetos heredan los métodos de abajo hacia arriba, es decir, siempre tienen preferencia los métodos sobreescritos. Java hace una búsqueda dinámica del método aplicable empezando por el más próximo al tipo y escalando sucesivamente los supertipos hasta encontrar un método con la denominación especificada.

Próxima entrega: CU00692B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

USO DE SUPER PARA LLAMAR A MÉTODOS DE SUPERCLASE. EJEMPLO.

En apartados anteriores del curso hemos trabajando con herencia, polimorfismo y sobreescritura de métodos. Hemos usado como clases de ejemplo las clases Persona, Profesor, ProfesorInterino y ProfesorTitular, donde Profesor hereda de Persona y los profesores interino y titular de Profesor.



¿Cómo imprimir un listado con todos los profesores (titulares e interinos) y todos sus datos? En los métodos de Profesor solo tenemos disponibles los campos de Profesor, pero no los campos de ProfesorTitular o de ProfesorInterino. A su vez, ProfesorTitular (o ProfesorInterino) no conoce los campos de Profesor porque son privados, y no queremos hacerlos públicos para no romper el encapsulamiento (principio de ocultación) de la clase.

Hemos visto cómo en los constructores de subclases usábamos la sintaxis `super()` en la primera línea para invocar al constructor de la superclase. Veremos ahora que esta otra sintaxis: `super.nombreDelMetodo(parámetros si los hay)`; tiene como efecto la invocación de un método de la superclase. Consideremos el ejemplo anterior. En el método `mostrarDatos()` de la clase ProfesorInterino escribimos lo siguiente:

```
//Ejemplo aprenderaprogramar.com
public void mostrarDatos() {
    super.mostrarDatos();
    System.out.println("Comienzo interinidad: " + FechaComienzoInterinidad.getTime().toString());
}
```

¿Qué hemos hecho? Dentro del método “hijo” hemos incluido una llamada al método “padre” que había perdido visibilidad debido a que lo habíamos sobreescrito. Al ejecutar el programa ahora obtenemos:

```
Se procede a mostrar los datos de los profesores existentes en el listín
Datos Profesor. Profesor de nombre: Juan Hernández García con Id de profesor: Prof 22-387-11
Datos Profesor. Profesor de nombre: José Luis Morales Pérez con Id de profesor: Unknown
Comienzo interinidad: Fri Nov 22 12:40:38 CET 2019
```

Gracias a la instrucción `super.mostrarDatos()` hemos incluido dentro de un método sobreescrito el método deseado de la superclase. La llamada a `super` dentro de un método puede ocurrir en cualquier lugar dentro del código. Equivale a la ejecución del método de la superclase y es opcional: la hacemos sólo cuando nos resulta necesaria o conveniente para poder acceder a un método sobreescrito.

EJERCICIO

Considera que estás desarrollando un programa Java donde trabajas con la superclase `Profesor` y la subclase `ProfesorEmerito`. Crea el código para estas clases que cumpla los requisitos que indicamos.

Como atributos de la superclase tendremos `nombre` (`String`), `edad` (`int`) y `añosConsolidados` (`int`). Un método de la superclase será `double obtenerSalarioBase ()` que obtendrá el salario base como $(725 + \text{el número de años consolidados multiplicado por } 33.25)$.

En la subclase se trabajará con el campo adicional `añosEmerito`. Queremos hacer lo siguiente: sobreescribir el método `obtenerSalarioBase ()` en la subclase para calcular el salario base del profesor emérito invocando mediante la palabra clave **super** al cálculo del salario base de `Profesor` y añadiéndole la cantidad de $(47.80 * \text{añosEmerito})$.

Para comprobar si tu código es correcto puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00693B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

MODIFICADORES DE ACCESO JAVA: PUBLIC, PRIVATE, PROTECTED.

Hasta ahora habíamos dicho que una subclase no tiene acceso a los campos de una superclase de acuerdo con el principio de **ocultación de la información**. Sin embargo, esto podría considerarse como demasiado restrictivo.



Decimos que podría considerarse demasiado restrictivo porque limita el acceso a una subclase como si se tratara de una clase cualquiera, cuando en realidad la relación de una superclase con una subclase es más estrecha que con una clase externa. Por ello en diferentes lenguajes, Java entre ellos, se usa un nivel de acceso intermedio que no es ni *public* ni *private*, sino algo intermedio que se denomina como “acceso protegido”, expresado con la palabra clave **protected**, que significa que las subclases sí pueden tener acceso al campo o método.

El modificador de acceso *protected* puede aplicarse a todos los miembros de una clase, es decir, tanto a campos como a métodos o constructores. En el caso de métodos o constructores protegidos, estos serán visibles/utilizables por las subclases y otras clases del mismo package. El acceso protegido suele aplicarse a métodos o constructores, pero preferiblemente no a campos, para evitar debilitar el encapsulamiento. En ocasiones puntuales sí resulta de interés declarar campos con acceso protegido.

La sintaxis para emplear esta palabra clave es análoga a la que usamos con las palabras *public* y *private*, con la salvedad de que *protected* suele usarse cuando se trabaja con herencia. Desde un objeto de una subclase podremos acceder o invocar un campo o método declarado como *protected*, pero no podemos acceder o invocar a campos o métodos privados de una superclase. Declara un campo de una clase como *protected* y en un test crea un objeto de la subclase y trata de acceder a ese campo con una invocación directa del tipo `interino43.IdProfesor = "54-DY-87"`.

Java admite una variante más en cuanto a modificadores de acceso: la omisión del mismo (no declarar ninguno de los modificadores *public*, *private* o *protected*). En la siguiente tabla puedes comparar los efectos de usar uno u otro tipo de declaración en cuanto a visibilidad de los campos o métodos:

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No

EJERCICIO

Considera que estás desarrollando un programa Java donde trabajas con la superclase Profesor y la subclase ProfesorEmerito. Crea el código para estas clases que cumpla los requisitos que indicamos.

Como atributos de la superclase tendremos nombre (String), edad (int) y añosConsolidados (int) declarados como **protected**.

En la subclase se trabajará con el campo adicional añosEmerito declarado como **private**.

Un método de la subclase será double obtenerSalarioBase () que obtendrá el salario base como $(925 + \text{añosConsolidados} * 33.25 + 47.80 * \text{añosEmerito})$.

Intenta acceder directamente al campo añosConsolidados desde la subclase (como si fuera un campo más de la subclase) para implementar este método. ¿Es posible sin utilizar una invocación a super ni un método get? ¿Qué ocurre si el atributo en la superclase lo declaras private?

Para comprobar si tus respuestas son correctas puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00694B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

SOBREESCRIBIR MÉTODOS EN JAVA. EJEMPLO: TOSTRING

Con lo expuesto en anteriores apartados del tutorial, ya sabemos que mientras no se hayan sobreescrito, los métodos de la superclase universal Object estarán disponibles para todos los objetos. Dentro de los métodos de la clase Object hay varios importantes, entre los que podemos citar los métodos *toString()* que devuelve un tipo String y *equals (Object obj)* que devuelve un tipo boolean.



Veamos primero el método *toString()*. El propósito de este método es **asociar a todo objeto un texto representativo**. Llamar a *toString()* sobre un objeto Integer producirá un resultado que es más o menos obvio: nos devuelve el entero asociado, solo que en forma de String. ¿Pero qué ocurre cuando invocamos el método sobre un objeto definido por nosotros? Este sería el caso de una invocación como *System.out.println ("Obtenemos " + profesor1.toString());*. En este caso, el resultado que obtenemos es del tipo: "Obtenemos Profesor@1de9ac4". El método efectivamente nos ha devuelto un String, ¿pero qué sentido tiene lo que nos ha devuelto? El resultado obtenido consta del nombre de la clase seguido de una cadena "extraña" que representa la dirección de memoria en que se encuentra el objeto. Este resultado en general es poco útil por lo que el método *toString()* es un método que habitualmente se sobreescribe al crear una clase. De hecho, la mayoría de las clases de la biblioteca estándar de Java sobreescriben el método *toString()*. Por otro lado, es frecuente que cuando un programador incluye métodos como imprimir..., mostrar..., listar..., etc. de alguna manera dentro de ellos se realicen llamadas al método *toString()* para evitar la repetición de código. También es frecuente que *toString()* aparezca en tareas de depuración ya que nos permite interpretar de forma "humana" los objetos. Supongamos que en nuestra clase Persona redefinimos *toString()* de la siguiente manera:

```
public String toString() { return nombre.concat(" ").concat(apellidos); }
```

En la clase profesor, que hereda de Persona, podríamos tener:

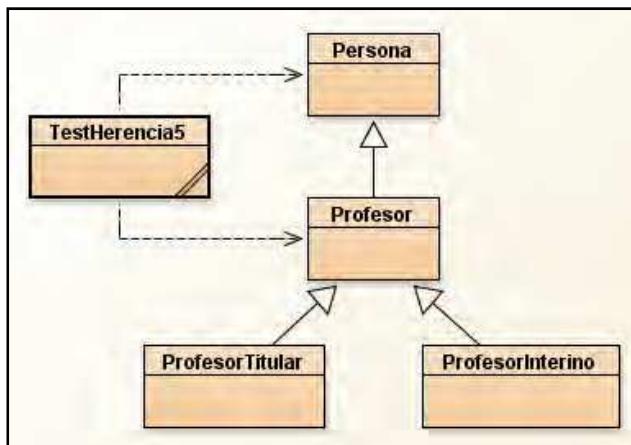
```
public String toString() { return super.toString().concat(" con Id de profesor: ").concat ( getIdProfesor() ); }
public void mostrarDatos() { System.out.println ("Los datos disponibles son: " + this.toString() ); }
```

En este ejemplo vemos, aparte del uso del método concat para concatenar Strings, una llamada a la superclase para recuperar el método *toString* de la superclase y cómo otro método (mostrarDatos) hace uso del método *toString()*. Usar *toString* es ventajoso porque no siempre nos interesa mostrar cadenas de texto por consolas en pantalla: tener los datos en un String nos permite p.ej. grabarlos en una base de datos, enviarlos en un correo electrónico, etc., además de poder mostrarlos por pantalla si queremos.

Aunque ya venimos usándolo, conviene remarcar que cuando usamos los métodos `println` y `print` del objeto `System.out`, cuando se incluye un término que no es un `String`, se invoca automáticamente el método `toString()` del objeto sin necesidad de escribirlo de forma explícita. Así `System.out.println(Profesor);` es equivalente a `System.out.println(Profesor.toString());`.

SOBREESCRIBIR MÉTODOS DE LA CLASE OBJECT: MÉTODO EQUALS

Ya hemos utilizado el método `equals` en diferentes ocasiones y sabemos que es la forma en que debemos comparar objetos. **Los objetos no se pueden comparar utilizando el operador ==.** El método `equals` está implementado en el API de Java para la mayoría de las clases. Por ello, podemos usarlo directamente para comparar `Strings` por ejemplo. Ahora bien, ¿qué ocurre en una clase creada por nosotros? Si escribimos algo como `if (profesor1.equals(profesor2))`, al no estar sobreescrito el método `equals` para la clase `Profesor`, el resultado es impredecible o incorrecto. Por tanto, para comparar objetos de tipo `Profesor` hemos de sobreescibir el método en la clase. Vamos a ver cómo realizaríamos la sobreescritura de este método dentro de la estructura de herencia en la que venimos trabajando. Escribe y compila el código que mostramos a continuación y trata de comprenderlo.



```
// Código que añadimos a la clase Persona. Sobreescritura del método equals ejemplo aprenderaprogramar.com
public boolean equals (Object obj) {
    if (obj instanceof Persona) {
        Persona tmpPersona = (Persona) obj;
        if (this.nombre.equals(tmpPersona.nombre) && this.apellidos.equals(tmpPersona.apellidos) &&
            this.edad == tmpPersona.edad) { return true; } else { return false; }
    } else { return false; }
} //Cierre del método equals
```

```
// Código que añadimos a la clase Profesor. Sobreescritura del método equals ejemplo aprenderaprogramar.com
public boolean equals (Object obj) {
    if (obj instanceof Profesor) {
        Profesor tmpProfesor = (Profesor) obj;
        if (super.equals(tmpProfesor) && this.idProfesor.equals(tmpProfesor.idProfesor) ) {
            return true; } else { return false; }
    } else { return false; }
} // Cierre del método equals
```

```
//Clase test herencia método equals ejemplo aprenderaprogramar.com
import java.util.Calendar;
public class TestHerencia5{
    public static void main (String [ ] Args) {
        Profesor profesor1 = new Profesor ("Juan", "Hernández García", 33);
        profesor1.setIdProfesor("Prof 22-387-11");
        Profesor profesor2 = new Profesor ("Juan", "Hernández García", 33);
        profesor2.setIdProfesor("Prof 22-387-11");
        Profesor profesor3 = new Profesor ("Juan", "Hernández García", 33);
        profesor3.setIdProfesor("Prof 11-285-22");

        Persona persona1 = new Persona ("José", "Hernández López", 28);
        Persona persona2 = new Persona ("José", "Hernández López", 28);
        Persona persona3 = new Persona ("Ramiro", "Suárez Rodríguez", 19);

        System.out.println ("¿Son iguales la persona1 y la persona2? " + persona1.equals(persona2));
        System.out.println ("¿Son el mismo objeto la persona1 y la persona2? " + (persona1 == persona2));
        System.out.println ("¿Son iguales la persona1 y la persona3? " + persona1.equals(persona3));

        System.out.println ("¿Son iguales el profesor1 y el profesor2? " + profesor1.equals(profesor2));
        System.out.println ("¿Son iguales el profesor1 y el profesor3? " + profesor1.equals(profesor3));

    } //Cierre del main
} //Cierre de la clase
```

Comprueba que obtienes estos resultados al ejecutar el main:

```
¿Son iguales la persona1 y la persona2? true
¿Son el mismo objeto la persona1 y la persona2? false
¿Son iguales la persona1 y la persona3? false
¿Son iguales el profesor1 y el profesor2? true
¿Son iguales el profesor1 y el profesor3? false
```

Analicemos ahora lo que hemos hecho. En la clase Persona, mediante la sobreescritura del método equals, hemos definido que para nosotros dos personas van a ser iguales si coinciden sus nombre, apellidos y edad. El criterio lo hemos fijado nosotros. Otra opción hubiera sido establecer que dos personas son iguales si coinciden nombre y apellidos. Fíjate que los Strings los comparamos usando el método equals del API de Java para los objetos de tipo String, mientras que la edad la comparamos con el operador == por ser un tipo primitivo. Otra cuestión relevante es el tratamiento de tipos: **el método equals requiere como parámetro un tipo Object** y no un tipo Persona. Así hemos de escribirlo para que realmente sea una redefinición del método de la clase Object. Si usáramos otra firma, no sería una redefinición o sobreescritura del método, sino un nuevo método. En primer lugar comprobamos si el objeto pasado como parámetro es un tipo Persona. Si no lo es, devolvemos como resultado del método false: los objetos no son iguales (no pueden serlo si ni siquiera coinciden sus tipos). En segundo lugar, una vez verificado que el objeto es portador de un tipo Persona, creamos una variable de tipo Persona a la que asignamos el objeto pasado como parámetro valiéndonos de casting (enmascaramiento). Esta variable la creamos para poder invocar campos y métodos de la clase Persona, ya que esto no podemos

hacerlo sobre un objeto de tipo Object. Con esta variable, realizamos las comparaciones oportunas y devolvemos un resultado.

En la clase Profesor hemos sobreescrito también el método equals con un tratamiento de tipos y uso de casting similar. En este caso invocamos el método equals de la superclase Persona, con lo que decimos que para que dos profesores sean iguales han de coincidir nombre, apellidos y edad. Además establecemos otro requisito para considerar a dos profesores iguales: que coincida su IdProfesor. Esto lo hacemos a nuestra conveniencia.

Finalmente en el test realizamos pruebas de los métodos implementados, comprobando por ejemplo que una persona1 puede ser igual a otra persona2 (de acuerdo con la definición dada al método equals) pero ambas ser diferentes objetos.

EJERCICIO

Define una clase Figura de la que hereden otras dos clases denominadas Cuadrado y Círculo. La clase figura debe tener al menos el campo dimensionPrincipal. Las clases Cuadrado y Círculo deben tener al menos un método calcularArea que permita calcular el área a partir de la dimensión principal, utilizando la fórmula matemática correspondiente. Además, sobreescribe el método equals para que dos cuadrados sean iguales si tienen igual dimensión principal, y dos círculos idem. A continuación crea un programa test donde crees varios círculos y cuadrados y hagas comprobaciones de igualdad usando el método equals.

Para comprobar si tu código es correcto puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00695B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CLASES Y MÉTODOS ABSTRACTOS EN JAVA. ABSTRACT CLASS.

Supongamos un esquema de herencia que consta de la clase Profesor de la que heredan ProfesorInterino y ProfesorTitular. Es posible que todo profesor haya de ser o bien ProfesorInterino o bien ProfesorTitular, es decir, que no vayan a existir instancias de la clase Profesor. Entonces, ¿qué sentido tendría tener una clase Profesor?



El sentido está en que una superclase permite unificar campos y métodos de las subclases, evitando la repetición de código y unificando procesos. Ahora bien, una clase de la que no se tiene intención de crear objetos, sino que únicamente sirve para unificar datos u operaciones de subclases, puede declararse de forma especial en Java: como clase abstracta. La declaración de que una clase es abstracta se hace con la sintaxis **public abstract class NombreDeLaClase { ... }**. Por ejemplo **public abstract class Profesor**. Cuando utilizamos esta sintaxis, no resulta posible instanciar la clase, es decir, no resulta posible crear objetos de ese tipo. Sin embargo, sigue funcionando como superclase de forma similar a como lo haría una superclase “normal”. La diferencia principal radica en que no se pueden crear objetos de esta clase.

Declarar una clase abstracta es distinto a tener una clase de la que no se crean objetos. En una clase abstracta, no existe la posibilidad. En una clase normal, existe la posibilidad de crearlos aunque no lo hagamos. El hecho de que no creemos instancias de una clase no es suficiente para que Java considere que una clase es abstracta. Para lograr esto hemos de declarar explícitamente la clase como abstracta mediante la sintaxis que hemos indicado. Si una clase no se declara usando *abstract* se cataloga como “clase concreta”. En inglés *abstract* significa “resumen”, por eso en algunos textos en castellano a las clases abstractas se les llama resúmenes. Una clase abstracta para Java es una clase de la que nunca se van a crear instancias: simplemente va a servir como superclase a otras clases. No se puede usar la palabra clave *new* aplicada a clases abstractas. En el menú contextual de la clase en BlueJ simplemente no aparece, y si intentamos crear objetos en el código nos saltará un error.

A su vez, las clases abstractas suelen contener métodos abstractos: la situación es la misma. Para que un método se considere abstracto ha de incluir en su signatura la palabra clave *abstract*. Además un método abstracto tiene estas peculiaridades:

- a) **No tiene cuerpo** (llaves): sólo consta de signatura con paréntesis.
- b) Su signatura **termina con un punto y coma**.

- c) **Sólo puede existir dentro de una clase abstracta.** De esta forma se evita que haya métodos que no se puedan ejecutar dentro de clases concretas. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
- d) Los métodos abstractos **forzosamente habrán de estar sobreescritos en las subclases.** Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta. Para que la subclase sea concreta habrá de implementar métodos sobreescritos para todos los métodos abstractos de sus superclases.

Un método abstracto para Java es un método que nunca va a ser ejecutado porque no tiene cuerpo. Simplemente, un método abstracto referencia a otros métodos de las subclases. ¿Qué utilidad tiene un método abstracto? Podemos ver un método abstracto como una palanca que fuerza dos cosas: la primera, que no se puedan crear objetos de una clase. La segunda, que todas las subclases sobreescriban el método declarado como abstracto.

Sintaxis tipo: *abstract public/private/protected TipodeRetorno/void (parámetros ...);*

Por ejemplo: *abstract public void generarNomina (int diasCotizados, boolean plusAntiguedad);*

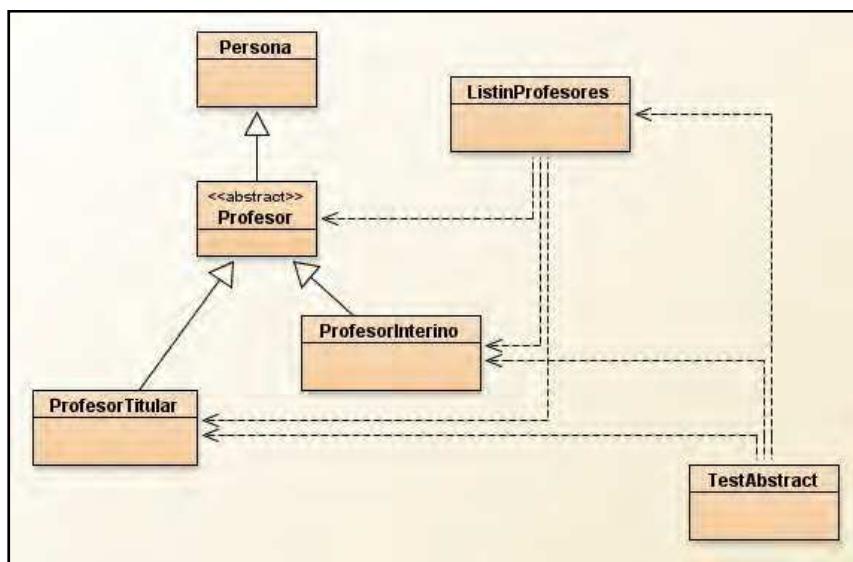
Que un método sea abstracto tiene otra implicación adicional: que podamos invocar el método abstracto sobre una variable de la superclase que apunta a un objeto de una subclase de modo que el método que se ejecute sea el correspondiente al tipo dinámico de la variable. En cierta manera, podríamos verlo como un método sobreescrito para que Java comprenda que debe buscar dinámicamente el método adecuado según la subclase a la que apunte la variable.

¿Es necesario que una clase que tiene uno o más métodos abstractos se defina como abstracta? Sí, si declaramos un método abstracto el compilador nos obliga a declarar la clase como abstracta porque si no lo hicieramos así tendríamos un método de una clase concreta no ejecutable, y eso no es admitido por Java.

¿Una clase se puede declarar como abstracta y no contener métodos abstractos? Sí, una clase puede ser declarada como abstracta y no contener métodos abstractos. En algunos casos la clase abstracta simplemente sirve para efectuar operaciones comunes a subclases sin necesidad de métodos abstractos. En otros casos sí se usarán los métodos abstractos para referenciar operaciones en la clase abstracta al contenido de la sobreescritura en las subclases.

¿Una clase que hereda de una clase abstracta puede ser no abstracta? Sí, de hecho esta es una de las razones de ser de las clases abstractas. Una clase abstracta no puede ser instanciada, pero pueden crearse subclases concretas sobre la base de una clase abstracta, y crear instancias de estas subclases. Para ello hay que heredar de la clase abstracta y anular los métodos abstractos, es decir, implementarlos.

Vamos a ver un ejemplo basado en el siguiente esquema:



En este diagrama de clases vemos cómo hemos definido una clase abstracta denominada Profesor. BlueJ la identifica señalando <>abstract<> en la parte superior del ícono de la clase. Sin embargo, hereda de la clase Persona que no es abstracta, lo cual significa que puede haber instancias de Persona pero no de Profesor.

El test que hemos diseñado se basa en lo siguiente: ProfesorTitular y ProfesorInterino son subclases de la clase abstracta Profesor. ListinProfesores sirve para crear un ArrayList de profesores que pueden ser tanto interinos como titulares y realizar operaciones con esos conjuntos. El listín se basa en el tipo estático Profesor, pero su contenido dinámico siempre será a base de instancias de ProfesorTitular o de ProfesorInterino ya que Profesor es una clase abstracta, no instanciable. En la clase de test creamos profesores interinos y profesores titulares y los vamos añadiendo a un listín. Posteriormente, invocamos el método imprimirListin, que se basa en los métodos `toString` de las subclases y de sus superclases mediante invocaciones sucesivas a super.

Por otro lado, en la clase ListinProfesores hemos definido el método `importeTotalNominaProfesorado()` que se basa en un bucle que calcula la nómina de todos los profesores que haya en el listín (sean interinos o titulares) mediante el uso de un método abstracto: `importeNomina()`. Este método está definido como `abstract public float importeNomina();` dentro de la clase abstracta profesor, e implementado en las clases ProfesorInterino y ProfesorTitular. El aspecto central de este ejemplo es comprobar cómo una clase abstracta como Profesor nos permite realizar operaciones conjuntas sobre varias clases, **ahormando código y ganando en claridad** para nuestros programas. Escribe este código:

```

public class Persona { //Código de la clase Persona ejemplo aprenderaprogramar.com
    private String nombre; private String apellidos; private int edad;
    public Persona() { nombre = ""; apellidos = ""; edad = 0; }
    public Persona (String nombre, String apellidos, int edad) { this.nombre = nombre; this.apellidos = apellidos; this.edad = edad; }
    public String getNombre() { return nombre; }
    public String getApellidos() { return apellidos; }
    public int getEdad() { return edad; }
    public String toString() { Integer datoEdad = edad;
        return "-Nombre: ".concat(nombre).concat(" -Apellidos: ").concat(apellidos).concat(" -Edad: ").concat(datoEdad.toString()); }
} //Cierre de la clase
  
```

En la clase Persona transformamos edad en un Integer para poder aplicarle el método `toString()`. De otra manera no podemos hacerlo por ser edad un tipo primitivo. Escribe este código:

```
public abstract class Profesor extends Persona {
    // Campo de la clase ejemplo aprenderaprogramar.com
    private String IdProfesor;

    // Constructores
    public Profesor () { super(); IdProfesor = "Unknown"; }
    public Profesor (String nombre, String apellidos, int edad, String id) { super(nombre, apellidos, edad); IdProfesor = id; }

    // Métodos
    public void setIdProfesor (String IdProfesor) { this.IdProfesor = IdProfesor; }
    public String getIdProfesor () { return IdProfesor; }
    public void mostrarDatos() {
        System.out.println ("Datos Profesor. Profesor de nombre: " + getNombre() + " " +
            getApellidos() + " con Id de profesor: " + getIdProfesor() );
    }
    public String toString () { return super.toString().concat("-IdProfesor: ").concat(IdProfesor); }
    abstract public float importeNomina (); // Método abstracto
} //Cierre de la clase
```

Hemos declarado la clase Profesor como abstracta. De hecho, tenemos un método abstracto (definido como *abstract* y sin cuerpo), lo cual de facto nos obliga a declarar la clase como abstracta. El método sobreescrito `toString` llama al método `toString` de la superclase y lo concatena con nuevas cadenas. Como clases que heredan de Profesor tenemos a ProfesorTitular y ProfesorInterino:

```
public class ProfesorTitular extends Profesor {
    // Constructor ejemplo aprenderaprogramar.com
    public ProfesorTitular(String nombre, String apellidos, int edad, String id) {
        super(nombre, apellidos, edad, id);
    }
    public float importeNomina () { return 30f * 43.20f; } //Método abstracto sobreescrito en esta clase
} //Cierre de la clase
```

```
import java.util.Calendar;

public class ProfesorInterino extends Profesor {
    // Campo de la clase ejemplo aprenderaprogramar.com
    private Calendar fechaComienzoInterinidad;

    // Constructores
    public ProfesorInterino (Calendar fechalinicioInterinidad) {
        super(); fechaComienzoInterinidad = fechalinicioInterinidad;
    }

    public ProfesorInterino (String nombre, String apellidos, int edad, String id, Calendar fechalinicioInterinidad) {
        super(nombre, apellidos, edad, id);
        fechaComienzoInterinidad = fechalinicioInterinidad;
    }
    public Calendar getFechaComienzoInterinidad () { return fechaComienzoInterinidad; } //Método
    public String toString () { //Sobrescritura del método
        return super.toString().concat (" Fecha comienzo interinidad: ").concat (fechaComienzoInterinidad.getTime().toString());
    }

    public float importeNomina () { return 30f * 35.60f; } //Método abstracto sobreescrito en esta clase
} //Cierre de la clase
```

```

import java.util.ArrayList; import java.util.Iterator;
public class ListinProfesores {
    private ArrayList <Profesor> listinProfesores; //Campo de la clase
    public ListinProfesores () { listinProfesores = new ArrayList <Profesor> (); } //Constructor
    public void addProfesor (Profesor profesor) { listinProfesores.add(profesor); } //Método
    public void imprimirListin() { //Método
        String tmpStr1 = ""; //String temporal que usamos como auxiliar
        System.out.println ("Se procede a mostrar los datos de los profesores existentes en el listín \n");
        for (Profesor tmp: listinProfesores) { System.out.println (tmp.toString () );
            if (tmp instanceof ProfesorInterino) { tmpStr1 = "Interino";} else { tmpStr1 = "Titular"; }
            System.out.println("-Tipo de este profesor:"+tmpStr1+" -Nómina de este profesor: "+(tmp.importeNomina())+ "\n");}
    } //Cierre método imprimirListin
    public float importeTotalNominaProfesorado() {
        float importeTotal = 0f; //Variable temporal que usamos como auxiliar
        Iterator<Profesor> it = listinProfesores.iterator();
        while (it.hasNext() ) { importeTotal = importeTotal + it.next().importeNomina(); }
        return importeTotal;
    } //Cierre del método importeTotalNominaProfesorado
} //Cierre de la clase ejemplo aprenderaprogramar.com

```

ProfesorTitular y ProfesorInterino se han definido como clases concretas que heredan de la clase abstracta Profesor. Ambas clases redefinen (obligatoriamente han de hacerlo) el método abstracto importeNomina() de la superclase. El método sobreescrito toString() de la clase ProfesorInterino llama al método toString() de la superclase y lo concatena con nuevas cadenas. El cálculo de importeNomina en ambas clases es una trivialidad: hemos incluido un cálculo sin mayor interés excepto que el de ver el funcionamiento de la implementación de métodos abstractos. ProfesorTitular lo hemos dejado con escaso contenido porque aquí lo usamos solo a modo de ejemplo de uso de clases abstractas y herencia. Su único cometido es mostrar que existe otra subclase de Profesor. Por otro lado, en la clase ListinProfesores tenemos un ejemplo de uso de instanceof para determinar qué tipo (ProfesorInterino o ProfesorTitular) es el que porta una variable Profesor. Iteramos con clase declarada Profesor y clases dinámicas ProfesorTitular y ProfesorInterino. Dinámicamente se determina de qué tipo es cada objeto y al invocar el método abstracto importeNomina() **Java determina si debe utilizar el método propio de un subtipo u otro.** En *imprimirListin* llegamos incluso a mostrar por pantalla de qué tipo es cada objeto usando la sentencia instanceof para determinarlo. Escribe y ejecuta el código del test:

```

import java.util.Calendar; //Ejemplo aprenderaprogramar.com
public class TestAbstract {
    public static void main (String [ ] Args) {
        Calendar fecha1 = Calendar.getInstance();
        fecha1.set(2019,10,22); //Los meses van de 0 a 11, luego 10 representa noviembre
        ProfesorInterino pi1 = new ProfesorInterino("José", "Hernández López", 45, "45221887-K", fecha1);
        ProfesorInterino pi2 = new ProfesorInterino("Andrés", "Moltó Parra", 87, "72332634-L", fecha1);
        ProfesorInterino pi3 = new ProfesorInterino ("José", "Ríos Mesa", 76, "34998128-M", fecha1);
        ProfesorTitular pt1 = new ProfesorTitular ("Juan", "Pérez Pérez", 23, "73-K");
        ProfesorTitular pt2 = new ProfesorTitular ("Alberto", "Centa Mota", 49, "88-L");
        ProfesorTitular pt3 = new ProfesorTitular ("Alberto", "Centa Mota", 49, "81-F");
        ListinProfesores listinProfesorado = new ListinProfesores ();
        listinProfesorado.addProfesor (pi1); listinProfesorado.addProfesor(pi2); listinProfesorado.addProfesor (pi3);
        listinProfesorado.addProfesor (pt1); listinProfesorado.addProfesor(pt2); listinProfesorado.addProfesor (pt3);
        listinProfesorado.imprimirListin();
        System.out.println ("El importe de las nóminas del profesorado que consta en el listín es " +
        listinProfesorado.importeTotalNominaProfesorado()+" euros");
    } } //Cierre del main y cierre de la clase

```

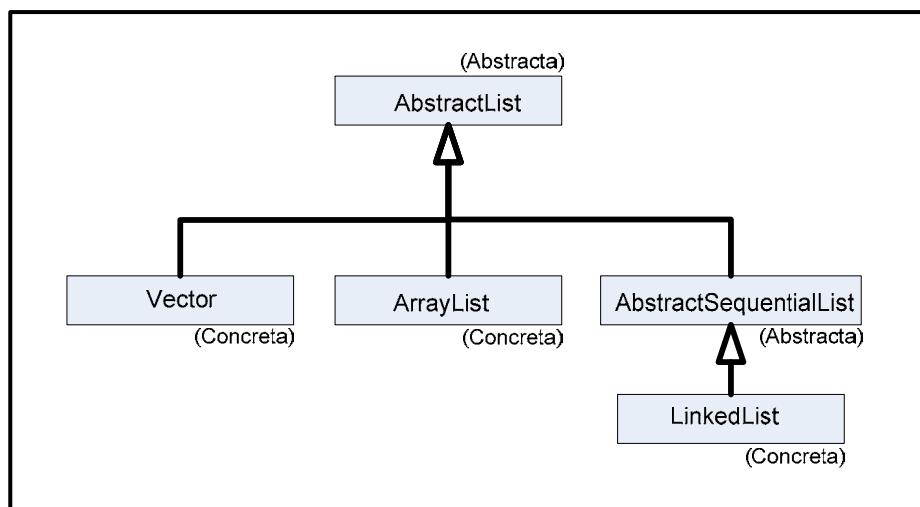
Comprueba el resultado de ejecución. El resultado del test nos muestra que operamos exitosamente sobre las dos clases usando abstracción:

```
Se procede a mostrar los datos de los profesores existentes en el listín
-Nombre: José -Apellidos: Hdez López -Edad: 45 -IdProfesor: 45221887-K Fecha comienzo interinidad: Fri Nov 22 11:55:28 CET 2019
-Tipo de este profesor: Interino -Nómina de este profesor: 1068.0
-Nombre: Andrés -Apellidos: Moltó Parra -Edad: 87 -IdProfesor: 72332634-L Fecha comienzo interinidad: Fri Nov 22 11:55:28 CET 2019
-Tipo de este profesor: Interino -Nómina de este profesor: 1068.0
-Nombre: José -Apellidos: Ríos Mesa -Edad: 76 -IdProfesor: 34998128-M Fecha comienzo interinidad: Fri Nov 22 11:55:28 CET 2019
-Tipo de este profesor: Interino -Nómina de este profesor: 1068.0
-Nombre: Juan -Apellidos: Pérez Pérez -Edad: 23 -IdProfesor: 73-K
-Tipo de este profesor: Titular -Nómina de este profesor: 1296.0
-Nombre: Alberto -Apellidos: Centa Mota -Edad: 49 -IdProfesor: 88-L
-Tipo de este profesor: Titular -Nómina de este profesor: 1296.0
-Nombre: Alberto -Apellidos: Centa Mota -Edad: 49 -IdProfesor: 81-F
-Tipo de este profesor: Titular -Nómina de este profesor: 1296.0

El importe de las nóminas del profesorado que consta en el listín es 7092.0 euros
```

CLASES ABSTRACTAS EN EL API DE JAVA

Java utiliza clases abstractas en el API de la misma forma que podemos nosotros usarlas en nuestros programas. Por ejemplo, la clase `AbstractList` del paquete `java.util` es una clase abstracta con tres subclases:



Como vemos, entre las subclases dos de ellas son concretas mientras que una todavía es abstracta. En una clase como `AbstractList` algunos métodos son abstractos, lo que obliga a que el método esté sobrescrito en las subclases, mientras que otros métodos no son abstractos.

Sobre un objeto de una subclase, llamar a un método puede dar lugar a:

- a) La ejecución del método **tal y como esté definido en la subclase**.
- b) La búsqueda del método **ascendiendo por las superclases** hasta que se encuentra y puede ser ejecutado. Es lo que ocurrirá por ejemplo con `toString()` si no está definido en la subclase.

EJERCICIO

Declara una clase abstracta `Legislador` que herede de la clase `Persona`, con un atributo `provinciaQueRepresenta` (tipo `String`) y otros atributos. Declara un método abstracto `getCamaraEnQueTrabaja`. Crea dos clases concretas que hereden de `Legislador`: la clase `Diputado` y la clase `Senador` que sobreescrivan los métodos abstractos necesarios. Crea una lista de legisladores y muestra por pantalla la cámara en que trabajan haciendo uso del polimorfismo.

Para comprobar si tu código es correcto puedes consultar en los foros aprenderaprogramar.com.

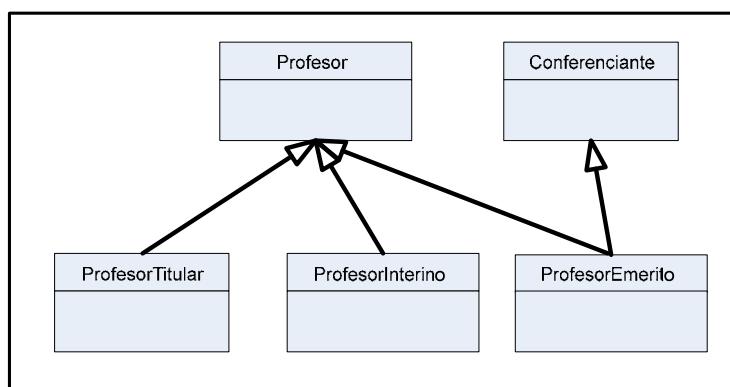
Próxima entrega: CU00696B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

CONCEPTO DE INTERFACE Y HERENCIA MÚLTIPLE EN JAVA. IMPLEMENTS.

En apartados anteriores del tutorial hemos estudiado los conceptos de herencia y polimorfismo. Hasta ahora hemos considerado escenarios en que una clase hereda solo de otra clase. ¿Sería posible plantear un escenario donde una clase hereda de más de una clase (herencia múltiple)?



El esquema de la figura anterior representaría que hubiera clases como ProfesorEmerito que heredarían de dos clases: Profesor y Conferencante. Esto sería un caso de herencia múltiple, y representaría que la subclase comparte las características de las dos superclases, y además tiene sus características específicas propias. La herencia múltiple, de cara a la consistencia de los programas y los lenguajes tiene una relativamente alta complejidad. De ahí que algunos lenguajes orientados a objetos la permitan y otros no. Java no permite la herencia múltiple, pero a cambio dispone de la construcción denominada “Interface” que permite una forma de simulación o implementación limitada de la herencia múltiple.

Ya hemos discutido el concepto de interfaz en alusión a la firma de métodos o la información pública de las clases. También hemos hecho una primera aproximación al término interface en Java, y a modo de símil dijimos que podía considerarse como una norma de urbanismo en una ciudad. Vamos a profundizar en el concepto de interface dentro de Java. Un interface es una construcción similar a una clase abstracta en Java, pero con las siguientes diferencias:

- En el encabezado se usa la **palabra clave interface** en lugar de class o abstract class. Por ejemplo `public interface NombreDeInterface {...}`
- Todo método es abstracto y público** sin necesidad de declararlo, es decir, no hace falta poner `abstract public` porque por defecto todos los métodos son `abstract public`. Por lo tanto un interface en Java no implementa ninguno de los métodos que declara: ninguno de sus métodos tiene cuerpo.
- Las interfaces **no tienen ningún constructor**.

- d) Un interfaz **solo admite campos de tipo “public static final”**, es decir, campos de clase, públicos y constantes. No hace falta incluir las palabras *public static final* porque todos los campos serán tratados como si llevaran estas palabras. Recordemos que *static* equivalía a “de clase” y *final* a “constante”. Las interfaces pueden ser un lugar interesante para declarar constantes que van a ser usadas por diferentes clases en nuestros programas.
- e) Una clase puede derivar de un interface de la misma manera en que puede derivar de otra clase. No obstante, se dice que **el interface se implementa (*implements*), no se extiende (*extends*)** por sus subclases. Por tanto para declarar la herencia de un interface se usa la palabra clave *implements* en lugar de *extends*.

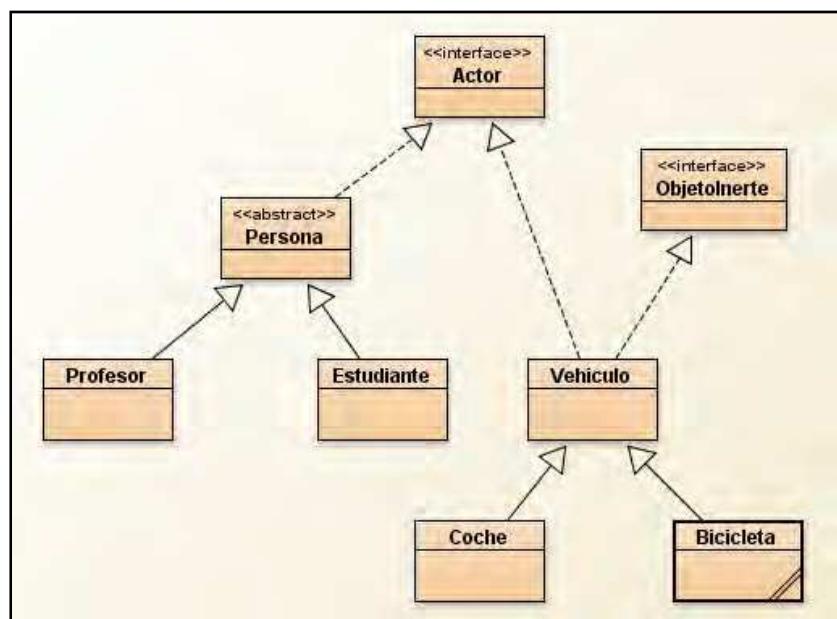
Una clase puede implementar uno o varios interfaces en Java (se indica con *extends NombreInterface1, NombreInterface2, ...etc.*), pero sólo puede extender una clase. Implementar varios interfaces en una sola clase es lo más parecido que tiene Java a la herencia múltiple.

Podemos declarar variables del tipo interfaz, pero para inicializarlas utilizaremos una clase concreta que lo implemente. Por ejemplo `List <String> miLista;` declara una variable con el tipo de la interface `List`. La inicialización `miLista = new List<String>();` no es posible porque no se puede crear un objeto del tipo definido por una interfaz. En cambio `miLista = new LinkedList<String>();` sí es válido.

Diremos que una interfaz en Java define un tipo cuyos métodos están todos sin implementar y que resulta equivalente a una herencia múltiple (de clases abstractas) en una clase. Si una clase implementa una interface, puede suceder:

- a) Que implemente los métodos de la interface sobreescribiéndolos (puede ser una clase concreta).
- b) Que no implemente los métodos de la interface: obligatoriamente será una clase abstracta y obligatoriamente ha de llevar la palabra clave *abstract* en su encabezado para así indicarlo.

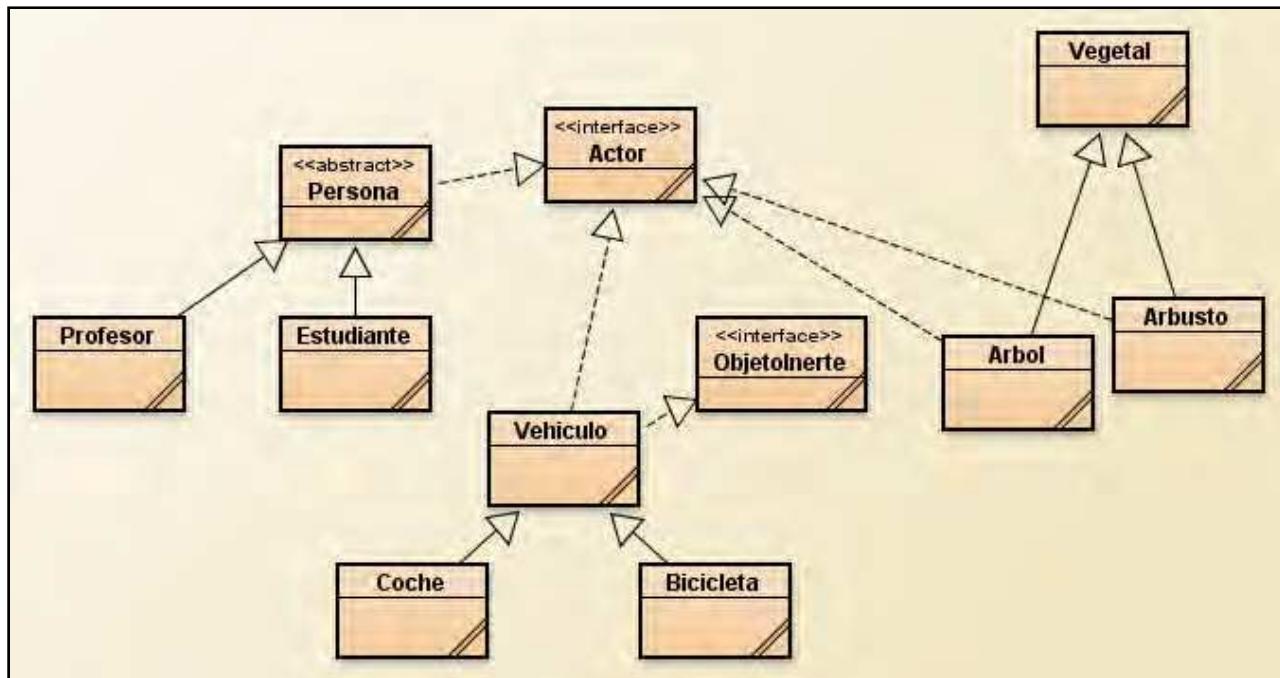
Consideremos un diagrama de clases como este, que podría emplearse para un programa de gestión en un centro educativo:



Vemos que las interfaces son identificadas por BlueJ con <<interface>> en la parte superior de su ícono. Dentro de estas clases las relaciones quedan determinadas por este código:

```
public interface Actor {...} : define la interfaz actor ejemplo aprenderaprogramar.com.
public abstract class Persona implements Actor {...} : define la clase abstracta Persona como implementación de la interfaz Actor.
public class Profesor extends Persona{...} : define la clase Profesor como extensión de la clase Persona.
public class Estudiante extends Persona{...} : define la clase Estudiante como extensión de la clase Persona.
public interface ObjetoInerte {...} : define la interfaz ObjetoInerte.
public class Vehiculo implements Actor, ObjetoInerte {...} : define que la clase Vehiculo implementa a dos interfaces, la interfaz Actor y la interfaz ObjetoInerte, es decir, que un vehículo es a la vez Actor y ObjetoInerte.
public class Coche extends Vehiculo {...} : define la clase Coche como extensión de la clase Vehiculo.
public class Bicicleta extends Vehiculo {...} : define la clase Bicicleta como extensión de la clase Vehiculo.
```

Una clase podría heredar de otra e implementar una o varias interfaces. En este caso en primer lugar se pone la relación de herencia respecto a la superclase y a continuación las interfaces que implementa. Por ejemplo en este esquema:



La definición de la clase Arbol sería así: `public class Arbol extends Vegetal implements Actor {...}`. La clase hereda de Vegetal e implementa a Actor.

¿Cómo saber si una clase es candidata a ser definida como una interfaz?

- Si necesitamos algún método con cuerpo ya sabemos que no va a ser una interfaz porque todos los métodos de una interfaz han de ser abstractos.
- Si necesitamos que una clase “herede” de más de una superclase, esas superclases son candidatas a ser interfaces.

- c) En algunos casos es igual de viable definir una clase como interfaz que como clase abstracta, pero puestos en esta situación preferiremos optar por una interfaz porque es más flexible y extensible: nos va a permitir que muchas clases implementen la interfaz (aprovechamos la herencia múltiple de las interfaces). En cambio, una clase abstracta sólo nos permite que una clase herede de ella.

Recordar que los campos declarados **son campos estáticos aunque no se indique específicamente**:

```
public interface Actor {  
    int activo = 1;  
    int inactivo = 0;  
    // ... resto del código de la interface ejemplo aprenderaprogramar.com  
}
```

En este caso *activo* e *inactivo* se comportan como *public static final* (constantes) para todas las clases que implementen esta interfaz.

¿Cuál es uno de los intereses principales de usar interfaces? Poder hacer uso del polimorfismo: por ejemplo poder reunir en una colección objetos del tipo interface pero que están implementados en distintas clases. O poder tratar en un bucle objetos de distintos tipos pero que pertenecen al mismo supertipo porque implementan una interfaz.

EJERCICIO

Responde a las siguientes preguntas:

- ¿Una clase puede heredar de dos clases en Java?
- ¿Una interface Java puede tener métodos que incluyan una sentencia while? ¿Una interface Java puede tener métodos que incluyan una sentencia System.out.println?
- ¿Un objeto Java puede ser del tipo definido por una interface? ¿Un objeto Java puede ser al mismo tiempo del tipo definido por una interface y del tipo definido por una clase que no implementa la interface? ¿Un objeto Java puede ser al mismo tiempo del tipo definido por una interface y del tipo definido por una clase que implementa la interface?

Para comprobar si tus respuestas son correctas puedes consultar en los foros aprenderaprogramar.com.

Próxima entrega: CU00697B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

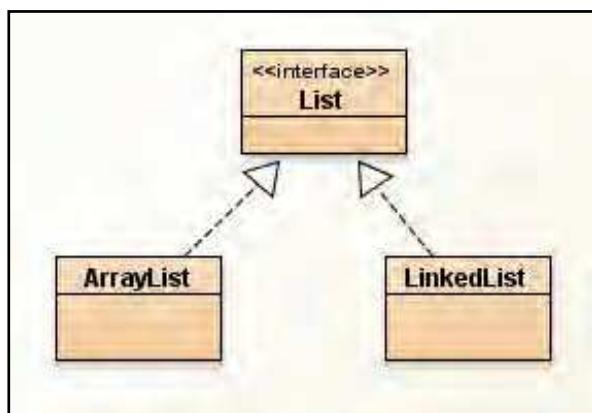
PARA QUÉ SIRVEN LAS INTERFACES EN JAVA

Si una interfaz define un tipo (al igual que una clase define un tipo) pero ese tipo no provee de ningún método podemos preguntarnos: ¿qué se gana con las interfaces en Java? La implementación (herencia) de una interfaz no podemos decir que evite la duplicidad de código o que favorezca la reutilización de código puesto que realmente no proveen código.



En cambio sí podemos decir que reúne las otras dos ventajas de la herencia: favorecer el mantenimiento y la extensión de las aplicaciones. ¿Por qué? Porque **al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos**. En el diagrama que vimos anteriormente tanto árboles como arbustos, vehículos y personas son de tipo Actor, de modo que podemos generar código que haga un tratamiento en común de todo lo que son actores. Por ejemplo, podemos necesitar una lista de Actores. Podemos declarar una variable como de tipo Actor (aunque no puedan existir instancias de Actor) que permita referenciar alternativamente a objetos de las distintas subclases de la interfaz.

Un aspecto fundamental de las interfaces en Java es hacer lo que ya hemos dicho que hace una interfaz de forma genérica: **separar la especificación de una clase (qué hace) de la implementación (cómo lo hace)**. Esto se ha comprobado que da lugar a programas más robustos y con menos errores. Pensemos en el API de Java. Por ejemplo, disponemos de la interfaz List que es implementada por las clases ArrayList y LinkedList (y también por otras varias clases).



El hecho de declarar una variable de tipo lista, por ejemplo List <String> miLista; nos dice que miLista va a ser una implementación de List, pero todavía no hemos definido cuál de las posibles implementaciones va a ser. De hecho, el código podría definir que se implementara de una u otra manera en función de las circunstancias usando condicionales. O a nivel de programación, mantendríamos la definición como List y nos permitiría comprobar el rendimiento de distintas

configuraciones (hacer funcionar miLista bien como ArrayList bien como LinkedList viendo su rendimiento). La variable declarada se crea cuando escribimos miLista = new LinkedList <String> (); o también se puede usar la sintaxis: List <String> miLista = new LinkedList <String> ();

Usar una u otra implementación puede dar lugar a diferentes rendimientos de un programa. ArrayList responde muy bien para la búsqueda de elementos situados en posiciones intermedias pero la inserción o eliminación de elementos puede ser más rápida con una LinkedList. Declarando las variables simplemente como List tendremos la posibilidad de que nuestro programa pase de usar un tipo de lista a otro tipo.

Como List es un tipo, podemos especificar los métodos para que requieran List y después enviarles como parámetro bien un ArrayList bien un LinkedList sin tener que preocuparnos de hacer cambios en el código en caso de que usáramos uno u otro tipo de lista. En esencia, usando siempre List, el único sitio donde habría que especificar la clase concreta sería donde se declara la creación de la variable, con lo cual todo nuestro código (excepto el lugar puntual donde se crea la variable) es independiente del tipo de lista que usemos y esto resulta ventajoso porque pasar de usar un tipo de lista a usar otro resultará muy sencillo.

Los métodos de ArrayList en algunos casos definen los métodos abstractos de List, y en otros casos son específicos. Recordar que en List **todos los métodos son abstractos por ser una interfaz**, aunque no se indique específicamente en la documentación del API de Java. Recordar también que List, por ser **una interfaz no tiene constructores y no es instanciable**. Al ver la documentación del API nos puede parecer una clase, pero la ausencia de constructor (aparte del propio nombre en el encabezado) delata que no se trata de una clase.

java.util

Interface List<E>

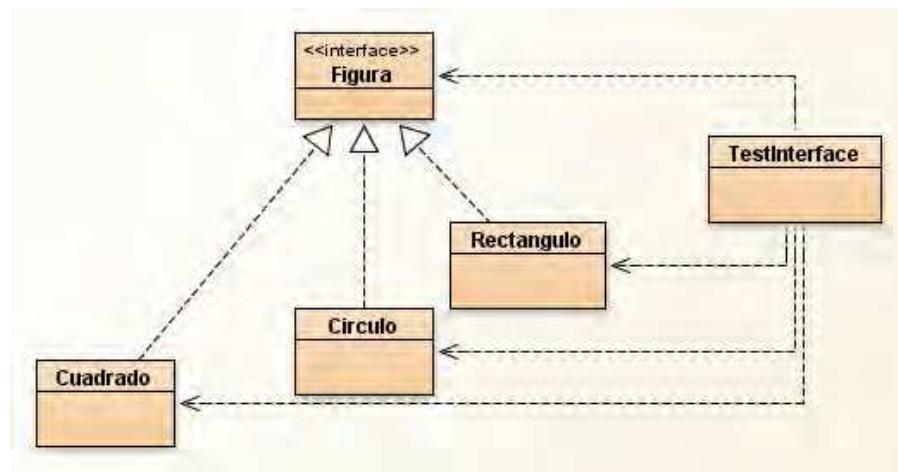
All Superinterfaces: [Collection<E>](#), [Iterable<E>](#)

All Known Implementing Classes: [AbstractList](#), [AbstractSequentialList](#), [ArrayList](#), [AttributeList](#), [CopyOnWriteArrayList](#), [LinkedList](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [Vector](#)

Otra ventaja clara de las interfaces es que nos permiten declarar constantes que van a estar disponibles para todas las clases que queramos (implementando esa interfaz). Nos ahorra código evitando tener que escribir las mismas declaraciones de constantes en diferentes clases.

EJEMPLO SENCILLO DE INTERFACE EN JAVA

Vamos a ver un ejemplo simple de definición y uso de interface en Java. Las clases que vamos a usar y sus relaciones se muestran en el esquema. Escribe el código y ejecútalo.



```

public interface Figura { // Ejemplo aprenderaprogramar.com
    float PI = 3.1416f; // Por defecto public static final. La f final indica que el número es float
    float area(); // Por defecto abstract public
} //Cierre de la interface
  
```

```

public class Cuadrado implements Figura { // La clase implementa la interface Figura
    private float lado;
    public Cuadrado (float lado) { this.lado = lado; }
    public float area() { return lado*lado; }
} //Cierre de la clase ejemplo aprenderaprogramar.com
  
```

```

public class Círculo implements Figura{ // La clase implementa la interface Figura
    private float diametro;
    public Círculo (float diametro) { this.diametro = diametro; }
    public float area() { return (PI*diametro*diametro/4f); }
} //Cierre de la clase ejemplo aprenderaprogramar.com
  
```

```

public class Rectángulo implements Figura{ // La clase implementa la interface Figura
    private float lado; private float altura;
    public Rectángulo (float lado, float altura) { this.lado = lado; this.altura = altura; }
    public float area() { return lado*altura; }
} //Cierre de la clase ejemplo aprenderaprogramar.com
  
```

```

import java.util.List; import java.util.ArrayList; //Test ejemplo aprenderaprogramar.com
public class TestInterface {
    public static void main (String [ ] Args) {
        Figura cuad1 = new Cuadrado (3.5f); Figura cuad2 = new Cuadrado (2.2f); Figura cuad3 = new Cuadrado (8.9f);
        Figura circ1 = new Círculo (3.5f); Figura circ2 = new Círculo (4f);
        Figura rect1 = new Rectángulo (2.25f, 2.55f); Figura rect2 = new Rectángulo (12f, 3f);
        List <Figura> serieDeFiguras = new ArrayList <Figura> ();
        serieDeFiguras.add (cuad1); serieDeFiguras.add (cuad2); serieDeFiguras.add (cuad3);
        serieDeFiguras.add (circ1); serieDeFiguras.add (circ2); serieDeFiguras.add (rect1); serieDeFiguras.add (rect2);
        float areaTotal = 0;
        for (Figura tmp: serieDeFiguras) { areaTotal = areaTotal + tmp.area(); }
        System.out.println ("Tenemos un total de " + serieDeFiguras.size() + " figuras y su área total es de " +
        areaTotal + " uds cuadradas") } } //Cierre del main y de la clase
  
```

El resultado de ejecución podría ser algo así:

```
Tenemos un total de 7 figuras y su área total es de 160.22504 uds cuadradas
```

En este ejemplo **comprobamos que la interface Figura define un tipo**. Podemos crear un ArrayList de figuras donde tenemos figuras de distintos tipos (cuadrados, círculos, rectángulos) aprovechándonos del polimorfismo. Esto nos permite darle un tratamiento común a todas las figuras. En concreto, usamos un bucle for-each para recorrer la lista de figuras y obtener un área total.

IMPLEMENTAR UNA INTERFACE DEL API JAVA. EJEMPLO.

El API de Java define interfaces que aparte de usarlas para definir tipos, nosotros podemos implementar en una clase propia en nuestro código. Esto tiene cierta similitud con hacer una redefinición de un método (ya hemos visto cómo redefinir métodos como `toString()`), pero no es exactamente lo mismo. Para empezar, algunos métodos como `toString()` están definidos en la clase Object. Estos métodos declarados en la clase Object los podemos redefinir en una clase propia sin necesidad de escribir nada en cabecera de la clase, puesto que por defecto todo objeto hereda de Object. Para utilizar interfaces, como la interfaz Comparable, habremos de escribir en cabecera de la clase:

```
public class NombreDeLaClase implements Comparable <NombreDeLaClase> { ... }
```

Por ejemplo `public class Persona implements Comparable <Persona>`.

¿Qué interés tiene implementar una interface del API si no nos proporciona código ninguno? Tal y como dijimos en su momento, una interface puede verse en relación a la programación como una norma urbanística en una ciudad. Si lees la documentación de la interfaz, aunque no proporciona código, sí proporciona instrucciones respecto a características comunes para las clases que la implementen y define qué métodos han de incluirse para cumplir con la interfaz y para qué servirán esos métodos. Si implementamos la interface, lo que hacemos es ajustarnos a la norma. Y si todos los programadores se ajustan a la misma norma, cuando un programador tiene que continuar un programa iniciado por otro no tiene que preguntarse: ¿qué método podré usar para comparar varios objetos de este tipo y ponerlos en orden? Y no hay que preguntárselo porque en general los programadores se ciñen a lo establecido por el API de Java: para comparar varios objetos y ponerlos en orden ("orden natural") se implementa la interfaz Comparable y su método `compareTo()`. Y además, ya sabemos qué tipo ha de devolver ese método y cómo ha de funcionar, porque así lo indica la documentación de la interface.

Muchas clases del API de Java ya tienen implementada la interface Comparable. Por ejemplo la clase Integer tiene implementada esta interfaz, lo que significa que el método `compareTo()` es un método disponible para cualquier objeto de tipo Integer.

No podemos conocer ni todas las clases ni todas las interfaces del API de Java. No obstante, a medida que vayamos realizando programas y adquiriendo práctica con Java, nos daremos cuenta de que algunas clases e interfaces son muy usadas. A base de usarlas, iremos memorizando poco a poco sus nombres y métodos. Otras clases o interfaces las usaremos ocasionalmente y recurrirremos a la consulta de documentación del API cada vez que vayamos a usarlas. Y otras clases o interfaces quizás no lleguemos a usarlas nunca.

EJERCICIO

Se plantea desarrollar un programa Java que permita representar la siguiente situación. Una instalación deportiva es un recinto delimitado donde se practican deportes, en Java interesa disponer de un método int getTipoDeInstalacion(). Un edificio es una construcción cubierta y en Java interesa disponer de un método double getSuperficieEdificio(). Un polideportivo es al mismo tiempo una instalación deportiva y un edificio; en Java interesa conocer la superficie que tiene y el nombre que tiene. Un edificio de oficinas es un edificio; en Java interesa conocer el número de oficinas que tiene.

Definir dos interfaces y una clase que implemente ambas interfaces para representar la situación anterior. En una clase test con el método main, crear un ArrayList que contenga tres polideportivos y dos edificios de oficinas y utilizando un iterator, recorrer la colección y mostrar los atributos de cada elemento. ¿Entre qué clases existe una relación que se asemeja a la herencia múltiple?

Puedes comprobar si tu respuesta es correcta consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00698B

Acceso al curso completo en aprenderaprogramar.com --> Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

RESUMEN DE HERENCIA EN JAVA

A lo largo del tutorial hemos intentado reflejar que la filosofía de la programación orientada a objetos, encarnada en la herencia, polimorfismo, encapsulamiento y ocultación de la información, etc. tiene una serie de ventajas que la hacen ser quizás la metodología de programación más utilizada hoy en día.



Resumimos como ventajas de la herencia:

1. **Evitar duplicidad y favorecer la reutilización de código** (las subclases utilizan el código de superclases).
2. **Facilitar el mantenimiento** de aplicaciones. Podemos cambiar las clases que usamos fácilmente.
3. **Facilitar la extensión** de las aplicaciones. Podemos crear nuevas clases a partir de otras existentes.

Con lo que hemos visto podemos decir que la herencia tiene dos implicaciones importantes: la primera, la reutilización de código (herencia de código) y la segunda el permitir el polimorfismo (herencia del tipo). La herencia admite tres variantes:

- a) **Herencia a partir de clases concretas** (*extends* o herencia simple): heredamos el tipo y la implementación. Opcionalmente en la clase que extiende agregaremos nuevos métodos o sobreescribiremos otros ya existentes.
- b) **Herencia a partir de interfaces** (*implements* o forma de herencia múltiple): heredamos el tipo sin implementación. Para poder instanciar, todos los métodos deben ser sobreescritos.
- c) **Herencia a partir de clases abstractas** (*extends* sobre una clase abstracta, variante de herencia simple): heredamos el tipo y posiblemente un fragmento de implementación. Para poder instanciar, aquellos métodos abstractos han de ser sobreescritos, y la subclase pasaría a ser concreta. Si no se implementan todos los métodos abstractos, la subclase sigue siendo abstracta.

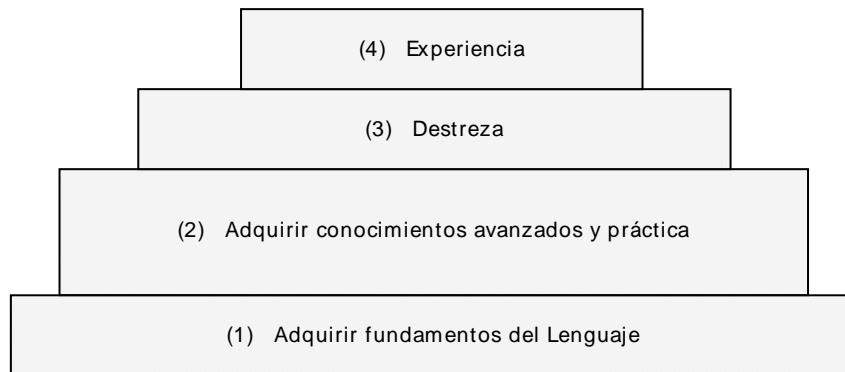
Observamos que en Java todos los tipos de herencia implican que se hereda el tipo. No en todos los lenguajes de programación ocurre igual.

Respecto a la herencia múltiple, Java no admite cualquier forma de herencia múltiple. Permite una forma de herencia múltiple de interfaces con la palabra clave *implements*, pero sólo herencia simple de clases, con la palabra clave *extends*.

PROGRESAR COMO PROGRAMADORES JAVA: SWING, GESTIÓN DE ERRORES Y MÁS ALLÁ. ¿QUÉ HEMOS APRENDIDO Y QUÉ NO HEMOS APRENDIDO CON ESTE TUTORIAL?

Hemos llegado al final del recorrido por Java en este tutorial que hemos denominado “Aprender programación Java desde cero”. ¿Qué hemos aprendido?

Desde nuestro punto de vista, los fundamentos para continuar progresando como programadores Java. Una de las mejores formas de abordar un área de conocimiento desconocida consiste en adquirir los fundamentos o principios fundamentales (cosa que hemos tratado de hacer mediante explicaciones y esquemas) y practicar (que en nuestro caso se traduciría por escribir código y hacer múltiples pruebas con el ordenador). Con la lectura y práctica de las secciones anteriores estamos en disposición de continuar progresando como programadores Java con unas bases sólidas. Esquemáticamente podríamos verlo así:



Nos hemos centrado en adquirir los fundamentos que nos permitan programar teniendo un entendimiento adecuado de lo que hacemos. Sin embargo, el recorrido para dominar Java, al igual que casi cualquier lenguaje, es un recorrido largo que puede requerir varios años.

Hemos hablado de la concepción del lenguaje, de la filosofía de la programación orientada a objetos, así como de algunas clases, interfaces y conceptos de diseño. Algunas clases importantes como `LinkedList`, `HashMap`, `Vector`, `HashSet`, `Collections`, y otras no las hemos abordado en profundidad o ni siquiera las hemos citado. No hemos discutido con amplitud el **diseño de clases** y programas (cohesión, acoplamiento, etc.) o su **prueba y depuración**. Tampoco hemos abordado las **clases internas**, de las que podemos decir brevemente que son clases definidas dentro de otra clase y cuya utilidad es definir un tipo que va a ser utilizado exclusivamente por la clase envolvente.

Tampoco hemos hablado de **gestión de errores** ni de las interfaces gráficas de usuario y de las importantes **bibliotecas del API de Java awt y swing**, o de la gestión de eventos, que constituyen partes muy importantes de la programación Java. No hemos abordado algunos patrones de diseño de interés como el patrón Singleton.

La extensión y contenidos de este curso básico sobre Java han sido motivo de discusión y los objetivos que planteamos el equipo editorial de aprenderaprogramar.com fueron los siguientes:

- a) Generar contenidos originales, didácticos y prácticos.
- b) Generar un curso breve, con contenidos claramente definidos y abordables en el marco de un curso o asignatura de duración limitada.
- c) Definir los fundamentos que permitieran la progresión natural de los alumnos mediante otros cursos o el entrenamiento individual.

Java es un lenguaje de relativa complejidad conceptual y de gran extensión, lo que hace ciertamente difícil su didáctica. Los tutoriales o cursos existentes muchas veces resultan demasiado áridos o extensos para los alumnos y esto se traduce en altas tasas de abandono. Si has llegado a esta parte final de este curso, confiamos en que sea porque te ha resultado didáctico, entretenido y de extensión adecuada. Si ha sido así, estamos seguros de que los fundamentos adquiridos serán sólidos y un valor añadido a tus capacidades en el área de la programación.

Próxima entrega: CU00699B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188

FINAL DEL CURSO TUTORIAL APRENDER A PROGRAMAR EN JAVA DESDE CERO

Con esta entrega llegamos al final del curso “Aprender programación Java desde cero”. Esperamos que haya sido un curso útil y ameno para todas las personas que lo hayan seguido. Y como en todo final, cabe hacer algunas consideraciones especiales.



Gracias al equipo humano de aprenderaprogramar.com que ha hecho posible su publicación, y en especial a Enrique González, Manuel Sierra, César Hernández, Javier Roa, Manuel Tello y Walter Sagástegui.

Gracias a todas las personas que de una u otra forma han participado enviando propuestas de mejora, comentarios, avisos de erratas, etc. y a los alumnos que han seguido el curso en la modalidad tutorizada on-line.

A todos los que no han participado siguiendo el curso a través de la web, desde aprenderaprogramar.com les agradecemos que nos hagan llegar una opinión o propuesta de mejora sobre el mismo, bien a través de correo electrónico a contacto@aprenderaprogramar.com, bien a través de los foros. Todas las opiniones son bienvenidas y nos sirven para mejorar.

A quienes hayan seguido el curso de forma gratuita y piensen que los contenidos son de calidad y que merece dar un pequeño apoyo económico para que se puedan seguir ofreciendo más y mejores contenidos en este sitio web, les estaremos muy agradecidos si realizan una pequeña aportación económica en forma de donación pulsando sobre el enlace que aparece en la página principal de aprenderaprogramar.com.

A quienes tengan interés en proseguir formándose en el área de programación con aprenderaprogramar.com, les remitimos a consultar la oferta formativa en http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=64&Itemid=87

Quienes quieran consultar toda la oferta de cursos que se ofrecen en aprenderaprogramar.com pueden hacerlo en la siguiente URL: http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=57&Itemid=86

Agradecemos que cualquier sugerencia, crítica, comentario o errata detectada que nos pueda permitir mejorar los contenidos para el futuro se nos haga llegar a la siguiente dirección de correo electrónico: contacto@aprenderaprogramar.com, o alternativamente a través de los foros aprenderaprogramar.com o del formulario de contacto que está a disposición de todos los usuarios en el portal web. A todos los que nos han leído y nos siguen, gracias. ¡Nos vemos en el próximo!

El autor y colaboradores

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188