Sumario

Paradigma Conceptos Avanzados

La palabra reservada static

Las clases poseen los modificadores de visibilidad ya enunciados para la declaración de sus elementos (privado, público, protegido y visibilidad de paquete). Sin embargo existen modificadores de visibilidad adicionales que afectan radicalmente el uso de un atributo, método o clase anidada. Uno de ellos es static.
Cuando un miembro de la clase (atributo o método) se declara static , indica que la visibilidad está dentro de la clase, no de una instancia de esta, por eso se los denomina métodos o atributos “de la clase” en lugar del objeto. Esta definición, si bien no es muy afortunada, tiene su origen en el hecho que no se necesita un objeto del tipo de la clase para acceder al elemento declarado como tal, pero se debe indicar como accederla a través de un nombre totalmente calificado por notación de punto que incluye el nombre de la clase en la que está definido.
Como no se necesita un objeto del tipo de la clase, indica claramente que estos elementos no pertenecerán al objeto que se crea a partir de la clase que los contiene y por lo tanto no podrán acceder tampoco a los miembros de un objeto simplemente “porque no los ve”. La razón de fondo para esto es que al indicar el modificador de visibilidad static se le indica al lenguaje que no asocie una referencia de tipo this al elemento, y, como se mencionó anteriormente, este es el medio por el cual se accede a los elementos miembro de un objeto en tiempo de ejecución.

Atributos static

Un atributo static se almacena en la memoria estática, por lo tanto, no se pueden crear atributos de este tipo cada vez que se crea un objeto.
Entonces, ¿cómo soluciona esta limitación el lenguaje? La respuesta en realidad es simple: se crea una sola vez el atributo en la memoria estática y dicho lugar de almacenamiento se compartirá por todos los objetos del mismo tipo, o, en otras palabras, todos los objetos del tipo de la clase en donde se declaró el atributo pueden leer y escribir en él. Más aún, si el atributo tiene un modificador de visibilidad que determine que éste sea visible (público, protegido o de paquete según sea el caso), con resolver el nombre totalmente calificado (paquete.nombreDeClase.nombreDeAtributo) podrá ser accedido por cualquier clase.

Métodos static

Se puede invocar un método static sin una instancia de la clase a la que pertenece análogamente como se lo hizo con el atributo. La razón de esto es que el método no tiene un this asociado, lo cual implica que no pertenece a la instancia de ningún objeto del tipo de la clase. Sin embargo, esta contenido por la clase e implica que se deberá resolver la visibilidad por medio de un nombre totalmente calificado con notación de punto para accederlo.

El patrón de diseño Singleton (instancia única)

Este patrón de diseño tiene la característica que sólo existirá un objeto de su tipo lo largo de toda la ejecución del código en el que se encuentre definido. Para lograr esta implementación en tecnología Java, se puede aprovechar la característica de los elementos estáticos de una clase.
Una declaración static significa que cuando el cargador de clases de la máquina virtual detecte el uso de un atributo de este tipo, el mismo será el primero en estar disponible en memoria, inclusive antes que se cree cualquier otro objeto. Por otra parte, los métodos static deben poder invocarse antes que cualquier otro método, por lo tanto la máquina virtual deja disponible la posibilidad de

accederlos al comienzo de cada programa que los incluya. Usando esta característica, se declara una clase con un formato particular que crea una instancia de sí misma.

```
public class InstanciaUnica {
      private static InstanciaUnica s = new InstanciaUnica();
      private InstanciaUnica() {}
      public static InstanciaUnica getS() {
            return s;
      }
      public void otroMetodo() { }
      public void otroMetodo2() { }
}
```

El secreto radica en los siguientes pasos:

1. Existe un atributo static en la clase InstanciaUnica y la máquina virtual lo pone en memoria antes que nada.
2. Como el atributo tiene una asignación, trata de resolverla para así terminar con la declaración.
3. Cuando trata de inicializar se encuentra con un operador new y lo resuelve.
4. El operador actúa sobre la misma clase InstanciaUnica , y, a pesar que el constructor de la clase InstanciaUnica es privado, lo ve por estar definido en la clase.
5. Como el hecho es tratar de dejar disponible un objeto del tipo InstanciaUnica , se debe poseer una forma de retornar la referencia en un método que pertenezca a la clase para que sea accesible en cualquier situación que pudiera existir, por lo tanto, no puede depender de la creación del objeto. Como este método es la única forma de obtener una referencia, siempre se accede por él y como retorna un atributo estático que no pertenece a un determinado objeto, se puede invocar al método “siempre” para obtener la referencia.
6. Por último, pero no menos importante, la visibilidad del constructor es privado, motivo por el cual el objeto sólo se puede crear desde dentro de la misma clase y nunca desde afuera, siendo la razón de por qué se crea una única instancia.

```
//Uso de un singleton
InstanciaUnica si = InstanciaUnica.getS();
si.otroMetodo();
si.otroMetodo2();
```

Se debe notar que una vez obtenida la referencia, se puede acceder a cualquier otro método definido dentro de la clase porque dicho objeto ya existe.

La palabra clave final.

Se puede declarar final antecediendo a la declaración de:

- Una variable
- Un método
- Una clase
- Constantes

Constantes.

Para crear una constante en Java se debe utilizar la palabra clave final antecediendo a su tipo. La declaración define una constante llamada PI cuyo valor son sólo los primeros cuatro decimales de dicho número irracional, 3.1416, y no puede ser cambiado.

```
final double PI = 3.1416;
```

Se debe recordar que por convención, los nombres de los valores constantes se escriben completamente en mayúsculas. Si un programa intenta cambiar el valor de una constante, el compilador muestra un mensaje de error.
Java permite además que una constante se inicie con un valor que, a partir de ese punto, se considerará constante sólo si el valor es asignado en el constructor o lo inicializa una variable estática.

```
public class Constantes {
	public final int VALOR_MAXIMO = 10;
	public final int VALOR_INICIAL;
	public Constantes(int i) {
		VALOR_INICIAL = i;
	}
}
```

Métodos finales

Se puede utilizar la palabra clave final en la declaración de método para indicar al compilador que este método no puede ser sobrescrito por las subclases.
Una subclase no puede sobrescribir métodos que hayan sido declarados como final en la superclase (por definición, los métodos finales no pueden ser sobrescritos). Si se intenta sobrescribir un método final, el compilador mostrará un mensaje de error no compilará el programa.
Una subclase tampoco pude sobrescribir métodos que se hayan declarado como static en la superclase. En otras palabras, una subclase no puede sobrescribir un método de clase.

```
public class Super {
	Number unNumero;
	public final void met(){ }
}

public class Sub extends Super {
	float unNumero;
	public Sub() {
		Number ej = super.unNumero;
	}
	public void met(){ // ERROR
	}
}
```

Se podría hacer que un método fuera final si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto a instanciar.

Clases finales

Se puede declarar que una clase sea final , lo que implica que la clase no puede tener subclases.

Se realiza por motivos de diseño o seguridad.

```
public final class ClaseFinal {
}
```

Cualquier intento posterior de crear una subclase de ClaseFinal resultará en un error del compilador.

Seguridad

Un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases parecen y sienten como la clase original pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada.
Para prevenir esta clase de modificación del código, se puede declarar que la clase sea final y así prevenir que se cree cualquier subclase de ella.
La clase String del paquete java.lang es una clase final sólo por esta razón. Esta es tan vital para la operación del compilador y del intérprete que Java debe garantizar que siempre que un método o un objeto utilicen un String , obtenga un objeto java.lang.String y no algún otro tipo derivado de String . Esto asegura que ningún String tendrá propiedades extrañas, inconsistentes o indeseables.

Si se intenta compilar una subclase de una clase final, el compilador mostrará un mensaje de error y no compilará el programa.

Diseño

Otra razón por la que se podría querer declarar una clase final son razones de diseño orientado a objetos. Se podría pensar que una clase es "perfecta" o que, conceptualmente hablando, la clase no debería tener subclases.

La clase Object

La clase Object es superclase de la cadena de Herencia de todas las clases Java, o sea, todas heredan de Object . Esto permite a los programas poder declarar referencias a Object y asignar cualquier tipo de objeto a la misma.
Como el lenguaje permite sólo asignar tipos de una subclase a uno de una superclase por conversiones explícitas de tipos (casting), esto permite a las declaraciones de referencias de tipo Object actuar como “puentes” en las asignaciones (se convierte cualquier clase a Object y luego se convierte nuevamente a su tipo de referencia original porque todas las clases “heredan” de Object). Esto da gran flexibilidad de programación, como por ejemplo, crear vectores de tipo Object y asignar cualquier clase a ellos. Sin embargo, se debe ser muy cuidadoso de no cometer errores al convertir nuevamente el objeto a su referencia original.
Para aclarar detalles importantes de la clase Object , se necesita avanzar con algunos temas. Por lo tanto sólo se presentará una introducción al mismo.

Método hashCode()

Un **HashCode** es un identificador de 32 bits que se almacena en un Hash en la instancia de la clase. Toda clase debe proveer de un método **hashCode()** que permite recuperar el Hash Code asignado, por defecto, por la clase Object. El HashCode tiene una especial importancia para el rendimiento de las tablas hash y otras estructuras de datos que agrupan objetos en base al cálculo de los HashCode.

Comparación de objetos

El operador == determina si dos referencias son iguales una a la otra, ya que los objetos se crean en el heap y en las variables del stack sólo se almacena la referencia al lugar de memoria en el que se encuentran. Por lo tanto queda por resolver todavía el problema de cómo comparar objetos.
Otra gran ventaja que se obtiene a partir que todas las clases heredan de Object es la de compartir sus métodos públicos. Basándose en esto, se puede resolver un problema muy común de la POO, el cual es responder a la pregunta ¿cuándo dos objetos son iguales? La respuesta es simple de concepto, pero complicada para implementarla en la codificación.

- Dos objetos son iguales cuando sus atributos significativos lo son.

Queda claro que el programador (y diseñador) deberá indicar la forma de comparar dos objetos ya que él decide que atributos son significativos para considerar a dos objetos iguales. Sin embargo resta encontrar el lugar para que esto se realice. En este punto llega al auxilio el hecho que Object es superclase de todas las clases en Java, por lo tanto se puede utilizar un método público en ella para realizar la comparación. Sin embargo, luego cada programador deberá sobrescribir dicho método poniendo en su interior la funcionalidad adecuada a su clase para determinar cuando los objetos del tipo de la clase son iguales.
Por lo tanto, esto brinda la ventaja que el programador decida cuando dos objetos son iguales, pero en contrapartida le delega la responsabilidad que realice la comparación. El método equals() brinda una manera de determinar si dos objetos son iguales a través de comparar sus atributos, ya que es un método de Object y se puede sobrescribir. Esto implica es el encargado de cumplir la acción antes descrita.
La implementación de equals() en Object utiliza == , por eso se debe sobrescribir para hacer algo distinto a comparar referencias. Si se sobrescribe equals , se deberá sobrescribir hashCode() porque se utilizan en conjunto por la máquina virtual para comparar e identificar unívocamente a un objeto (esto se comprenderá mejor cuando se explique la utilización de colecciones).
En general, si se remplaza uno de estos métodos, es necesario remplazar ambos, ya que existen importantes relaciones entre ellos que deben ser mantenidas. En particular, si dos objetos son iguales de acuerdo con el método equals , deben retornar el mismo valor hashCode (aunque la inversa no es cierta en general).
La forma en la cual equals determina la igualdad de dos objetos se dejan para el implementador, porque la definición de lo que es igual para dos objetos del mismo tipo es parte del trabajo de diseño para esa clase.

```
//implementación por defecto de .equals() declarada en Object
public boolean equals(Object obj) {
      return (this == obj);
}
```

Bajo esta implementación por defecto, dos referencias sólo son iguales si se refieren exactamente al mismo objeto. Del mismo modo, el valor por defecto de hashCode proporcionado por Object se calcula mediante la asignación de la dirección de memoria del objeto en un valor entero. Debido a que en algunas arquitecturas el espacio de direcciones es mayor que el rango de valores para int, es posible que dos objetos distintos puedan tener el mismo hashCode . Si se remplaza hashCode, todavía se puede utilizar el método System.identityHashCode para acceder a dicho valor por defecto. Como este valor es propio del sistema operativo en el que se encuentra ejecutándose un determinado programa, esta generado por un método nativo (es un método que se implementa para la plataforma en particular y es dependiente de ella.

```
//Declaración de hashcode en Object
public native int hashCode();
```

El modificador **native**, sólo puede asignarse a métodos. Indica que el método está escrito en un lenguaje dependiente de la plataforma, habitualmente en C. Cuando usemos un método nativo, en nuestro fichero .javasólo incluiremos la cabecera del método con el modificador **native** y terminado en punto y coma (;) (como si fuera un método abstracto)
Necesitaremos por tanto los ficheros .h y .c correspondientes que serán los que contengan el código fuente del método nativo. Estos ficheros podemos generarlos con el comando **javah**. Este comando nos generará los ficheros que necesitamos y creará los vínculos necesarios con nuestra clase java.
Por supuesto los ficheros todavía no contienen el código fuente, por tanto, el siguiente paso consiste en "rellenar" esos ficheros .h y .c con nuestro código nativo.

Una implementación basada en la igualdad definida en equals y hashCode es un valor por defecto sensible, pero para algunas clases, es conveniente delegar la implementación de como evaluar la igualdad a otro elemento que la defina. Por ejemplo, la clase Integer declara equals de la siguiente forma:

```
public boolean equals(Object obj) {
  return(obj instanceof Integer && intValue()==((Integer)obj).intValue());
}
```

Bajo esta definición, dos objetos Integer sólo son iguales si contienen el mismo valor entero.
Se debe cumplir con ciertos requisitos para que los programas funcionen correctamente, las cuales se denominan relaciones de equivalencia.

Reglas a cumplir para que dos objetos sean iguales.

Reflexiva: Para todo valor de referencia x , x.equals(x) debe retornar true.
Simétrica: Para todo valor de referencias x e y , x.equals(y) debe retornar true si también lo hace y.equals(x).
Transitiva: Para todo valor de referencias x , y , z , si x.equals(y) retorna true y también lo hace y.equals(z) , entonces x.equals(z) debe retornar true.
Consistente: Para todo valor de referencias x e y , múltiples invocaciones retornaran consistentemente true o false y nunca se alternará entre los resultados.
No nulificable: Para todo valor de referencia x , x.equals(null) debe retornar siempre false.

```
//Definición por rescritura de la igualdad de objetos del mismo tipo
public class Empleado {
	private String nombre;
	private static final double SALARIO_BASE = 1500.00;
	private double salario = 1500.00;
	private Fecha fechaNacimiento;
	public Empleado(String nombre, double salario, Fecha fDN) {
		this.nombre = nombre;
		this.salario = salario;
		this.fechaNacimiento = fDN;
	}
	public Empleado(String nombre, double salario) {
		this(nombre, salario, null);
	}
	public Empleado(String nombre, Fecha fDN) {
		this(nombre, SALARIO_BASE, fDN);
	}
	public Empleado(String nombre) {
		this(nombre, SALARIO_BASE);
	}
```

```
	public int hashCode() {
		final int prime = 31;
			int result = 1;
			result = prime * result + ((fechaNacimiento == null) ? 0 :
				fechaNacimiento.hashCode());
			result = prime * result + ((nombre == null) ? 0 :
				nombre.hashCode());
		long temp = Double.doubleToLongBits(salario);
		return prime * result + (int) (temp ^ (temp >>> 32));
	}
	public boolean equals(Object obj) {
		if (this == obj) return true;
		if (obj == null) return false;
		if (getClass() != obj.getClass()) return false;
		Empleado other = (Empleado) obj;
		if (fechaNacimiento == null) {
			if (other.fechaNacimiento != null) return false;
			} else if (!fechaNacimiento.equals(other.fechaNacimiento))
				return false;
			if (nombre == null) {
				if (other.nombre != null) return false;
			} else if (!nombre.equals(other.nombre)) return false;
			if (Double.doubleToLongBits(salario) != Double
					.doubleToLongBits(other.salario))
			return false;
		return true;
	}
}
```

El Método toString()

Otra de las ventajas que ofrece el hecho que todas las clases hereden de Object es el acceso a uno de sus métodos de utilidad: toString . Este retorna un String que identifica al objeto y que será acorde a la implementación realizada de la clase que lo define (en otras palabras, como cada clase lo puede sobrescribir, cada una puede implementar su versión del método). Por ejemplo, muchas clases retornan el contenido de sus atributos. Otras retornan información del objeto o crean mensajes descriptivos.

Como en varios casos retornan contenidos de variables, se utiliza para concatenar cadenas ya que existen métodos de utilidad que invocan automáticamente a este método. Notoriamente, el método System.out.println está sobrecargado y entre otras cosas recibe un Object como argumento. En su interior, System.out.println invoca a toString cuando recibe como argumento la referencia a un objeto.

Como el método está declarado en Object y se puede escribir para adecuarlo a la clase que se cree, cuando se sobrescribe toString se oculta la visibilidad del método de Object y se invoca al sobrescrito en la subclase.

Un caso especial y muy utilizado es el de las clases de envoltorio de los tipos primitivos, las cuales sobrescriben el método para convertir el valor almacenado a String .

```
public String toString() {
	return "Empleado [nombre=" + nombre + ", salario=" + salario
		+ ", fechaNacimiento="  + fechaNacimiento + "]";
}
```

Palabra clave Abstract

Clases abstractas

Una clase abstracta modela un tipo de objetos donde algunos métodos deben ser especificados en la subclase porque su funcionamiento en inherente a ellas.
Algunas veces, una clase que se ha definido representa un concepto abstracto y como tal, no debe ser implementado en ese momento.
En la programación orientada a objetos, se podría modelar conceptos abstractos pero no querer que se creen implementaciones de ellos. Por ejemplo, la clase Number del paquete java.lang representa el concepto abstracto de número. Tiene sentido modelar números en un programa, pero no tiene sentido crear un objeto genérico de números. En su lugar, la clase Number sólo tiene sentido como superclase de otras clases como Integer y Float que implementan números de tipos específicos. Las clases como Number , que implementan conceptos abstractos y no deben ser implementadas (no se deben crear objetos de su tipo), son llamadas clases abstractas. Una clase abstracta es una clase que sólo puede tener subclases, no puede ser instanciada.
Para declarar que una clase es una clase abstracta, se utiliza la palabra clave abstract en la declaración de la clase.

```
abstract class Number {
      . . .
}
```

Si se intenta crear un objeto del tipo de la clase abstracta, el compilador mostrará un error y no compilará el programa.

Métodos Abstractos

Una clase abstracta puede contener métodos abstractos, esto es, métodos que no tienen implementación. De esta forma, una clase abstracta puede definir un interfaz de programación para aplicaciones completa, incluso proporciona a sus subclases la declaración de todos los métodos necesarios para implementar dicha interfaz de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.
Una clase abstracta en la cual sólo se definen métodos abstractos se denomina clase abstracta pura.
Para mostrar un ejemplo de cuando sería necesario crear una clase abstracta con métodos abstractos se puede considerar que en una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc... Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Se pueden aprovechar esas similitudes y declararlas todas a partir de una misma superclase, como ser ObjetoGrafico .
Sin embargo, los objetos gráficos también tienen diferencias substanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos. Por otro lado, todos los ObjetoGrafico deben saber cómo dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.
Primero se debe declarar una clase abstracta, ObjetoGrafico , para proporcionar las variables miembro y los métodos que van a ser compartidos por todas las subclases, como la posición actual y el método moverA() .
También se deberían declarar métodos abstractos como dibujar() , que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase).

```
abstract class ObjetoGrafico {
      int x, y;
```

```
        . . .
        void moverA(int nuevaX, int nuevaY) {
              . . .
        }
        abstract void dibujar();
  }
```

Una clase abstracta no necesita contener un método abstracto. Pero todas las clases que contengan un método abstracto, o no proporcionen implementación para cualquier método abstracto declarado en su superclase, debe ser declarada como una clase abstracta obligatoriamente. Visto de otra manera, una subclase que hereda de una superclase se sigue considerando abstracta hasta que todos los métodos abstractos en ella dejen de serlo porque tiene al menos un servicio incompleto.

Polimorfismo

Definición

El polimorfismo se manifiesta cuando se conoce la operación a realizar pero se desconoce cuándo se diseña un programa que objeto la va a llevar a cabo. En Java esto se produce cuando se invoca a un método conocido luego de asignar la referencia al objeto que se desea ejecute la acción.
Cuando una acción (método) depende del objeto que la ejecuta, se lo denomina polimorfismo, porque tiene muchas “formas” de ejecutarse. Cada forma de ejecutarse estará determinada por el comportamiento del objeto en el cual fue implementado el método.
Una referencia puede serlo de distintos objetos, pero si la acción en todos ellos es la misma, cambiando la referencia se indica sobre que objeto se ejecutará la acción, por lo tanto, el hecho de conocer la acción permite valerse de una variable a la que se le asignará la referencia al objeto en particular que se quiera invocar, para luego utilizar con notación de punto la invocación al mismo.

Invocación virtual de métodos

Para realizar una invocación virtual de métodos (virtual porque en tiempo de diseño no se conoce a que objeto pertenece el método a ejecutar) sólo se necesita asignar la referencia al objeto que tiene en método a ejecutar. Esto si bien brinda un gran poder de ejecución (la referencia “virtual” se resuelve en tiempo de ejecución en el momento de asignarle valor a la variable que se utilizará para realizar la invocación del método) tiene ciertas limitaciones que deben ser tenidas en cuenta al momento de programar.
Por ejemplo, cuando se declara una referencia, se debe recordar que está permitido asignar valores de subclases a referencias de superclases, pero no a la inversa (al menos, no sin hacer una conversión de tipo explícita). Por lo tanto, si este es el caso, sólo se podrá invocar a los métodos que “ve” la variable de referencia. En otras palabras, sólo se podrán acceder a métodos que estén declarados en la superclase y rescritos en las subclases.
Cualquier intento de acceder a un elemento de la subclase que no esté definido en la superclase será un error en tiempo de compilación.

Annotation

Introducción

Las anotaciones en Java se incluyeron a partir de Java 5. Éstas son metadatos que se pueden asociar a clases, miembros, métodos o parámetros. Nos dan información sobre el elemento que tiene la anotación y permiten definir cómo queremos que sea tratado por el framework de ejecución.

Además, las anotaciones no afectan directamente a la semántica del programa, pero sí a la forma en que los programas son tratados por las herramientas y librerías.
Pero no sólo disponemos de las anotaciones por defecto de Java, si no que podemos crear las nuestras propias y así disponer de metadatos útiles en tiempo de ejecución.

Creando la anotación

En Java, una anotación se define por medio de la palabra reservada ***@interface***. Hay que tener ciertas consideraciones en cuenta a la hora de crearlas:

- Cada método dentro de la anotación es un elemento.
- Los métodos no deben tener parámetros o cláusulas *throws*.
- Los tipos de retorno están restringidos a tipos primitivos, String, Class, enum, anotaciones, y arrrays de los tipos anteriores.
- Los métodos pueden tener valores por defecto

Vamos a crear una anotación que indique cuántas veces debe ejecutarse cada elemento que se anote con ella:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MultipleInvocation {
    int timesToInvoke() default 1;
}
```

Como se ve en el código, hemos usado algunas meta-anotaciones para definir ciertos parámetors en nuestra anotación. Veamos cuáles de estas se pueden usar al crear nuestras anotaciones y qué significan:

- ***@Target*** – Especifica el tipo de elemento al que se va a asociar la anotación.
    - ElementType.*TYPE* – se puede aplicar a cualquier elemento de la clase.
    - ElementType.*FIELD* – se puede aplicar a un miebro de la clase.
    - ElementType.*METHOD* – se puede aplicar a un método
    - ElementType.*PARAMETER* – se puede aplicar a parámetros de un método.
    - ElementType.*CONSTRUCTOR* – se puede aplicar a constructores
    - ElementType.*LOCAL_VARIABLE* – se puede aplicar a variables locales
    - ElementType.*ANNOTATION_TYPE* – indica que el tipo declarado en sí es un tipo de anotación.
- ***@Retention*** – Especifica el nivel de retención de la anotación (cuándo se puede acceder a ella).
    - RetentionPolicy.*SOURCE* — Retenida sólo a nivel de código; ignorada por el compilador.
    - RetentionPolicy.*CLASS* — Retenida en tiempo de compilación, pero ignorada en tiempo de ejecución.
    - RetentionPolicy.*RUNTIME* — Retenida en tiempo de ejecución, sólo se puede acceder a ella en este tiempo.
- ***@Documented*** – Hará que la anotación se mencione en el javadoc.
- ***@Inherited*** – Indica que la anotación será heredada automáticamente.

En nuestro caso hemos dicho que sea en tiempo de ejecución y que se aplique a métodos. También le hemos añadido un valor por defecto con la palabra ***default***

Usando la anotación

El funcionamiento de uso es muy sencillo: basta etiquetar con la anotación los métodos que queramos:

```
@MultipleInvocation
public void singleFire() {
    ammo--;
    System.out.println("Single fire! Ammo left: " + ammo);
}
```

Leyendo la anotación durante la ejecución.

La lectura de la anotación en tiempo de ejecución se realiza mediante reflexión.

Annotations comunes

En el JDK 5 se encuentran tres anotaciones que pueden ser utilizadas directamente :

Anotación	Función
Override	Utilizado para indicar que el método ha modificado su comportamiento ("override") sobre su superclase.
Deprecated	Utilizado para indicar que el uso de determinado elemento no es recomendable o ha dejado de ser actualizado.
SuppressWarnings	Permite suprimir mensajes del compilador relacionados con advertencias/avisos.

Interfaces

Definición

Una interfaz pública (en realidad este es su único posible valor ya que no existen interfaces de otra clase de visibilidad) es un convenio con la clase que la implementa, la cual deberá escribir el código que se ejecutará en el método cuyo prototipo se declara en la interfaz.
Las interfaces admiten dos tipos de declaraciones: métodos públicos sin bloque de sentencias asociado (iguales a los métodos abstractos) y constantes.
Este convenio es el que permite, a través de una referencia en común entre dos clases que la implementan, realizar llamados virtuales a métodos. La referencia en común es justamente del tipo de la interfaz (se pueden crear variables de referencias del tipo de una interfaz, lo que no se puede hacer es crear objetos de su tipo porque los métodos no tienen bloques de sentencias asociados, es decir, son abstractos a pesar de no existir una declaración explícita en el código de este hecho).
Por lo tanto, la interfaz de Java es la declaración formal del contrato, de manera que en ella sólo se encuentran los prototipos de los métodos a implementar en la clase y por ser estos “abstractos“, la clase que implemente la interfaz tiene obligación de rescribirlos o no se podrán crear objetos de su tipo (debe cumplir con el “contrato”).
Muchas clases no relacionadas pueden implementar una interfaz y de esta manera utilizarla como “puente de referencias” para llamar las acciones con el mismo nombre en clases que no estén en la misma cadena de herencia, pero que implementen la misma interfaz.
Una clase puede implementar muchas interfaces, la única salvedad es que para realizar esto cada declaración deberá separarse con comas.

La sintaxis es:

```
< declaración_clase> ::=< modificador> class < nombre>
      [extends < superclase>] [implements < interfaz> [,< interfaz >]* ] {

      < declaraciones>*

}
```

Declaran métodos que se esperan que una o más clases los implementen, o, en otras palabras, que las clases que implementen la interfaz provean el cuerpo del método o bloque de sentencias asociado.
Determinan la interfaz del objeto cuya clase la implemente sin revelar el cuerpo de la clase misma. Con sólo conocer los métodos que existen en la interfaz y las clases que la implementan, es suficiente para realizar invocaciones de los mismos en objetos del tipo de dichas clases y no es necesario brindar ninguna otra información de los servicios que prestan los objetos del tipo de las clases que implementan la interfaz, mucho menos aún de su estructura interna.
Capturan similitudes entre clases no relacionadas sin forzar relaciones de asociación, agregación, composición o herencia entre ellas, o, mejor dicho, cuando dos clases realizan las mismas acciones pero pertenecen a cadenas de herencia diferente, pueden compartir invocaciones virtuales de métodos si implementan la misma interfaz y luego valerse de declarar una variable del tipo de la interfaz y asignarle las referencias a los objetos de los que se quiera utilizar sus servicios.

Simula la herencia múltiple porque se pueden implementar muchas interfaces. Si se condiciona a que sólo se puede hacer herencia múltiple si se extiende una sola clase concreta y tantas clases abstractas puras (clases que no poseen ningún método concreto) como se desee, se estaría frente al caso de la herencia simple e interfaces de Java.

Polimorfismo usando interfaces

Para comprender como las interfaces pueden relacionar cadenas de herencia totalmente disímiles pero que comparten acciones en común, se plantea un nuevo conjunto de clases que pertenecen a cadenas de herencia que aparentemente no tiene nada en común y son totalmente diferentes de los ejemplos anteriores. El punto es demostrar como las interfaces consiguen formar verdaderos “puentes” entre clases para que se pueda utilizar comportamiento en común de los servicios que proveen.

```
Set set;
set=new TreeSet();
set=new LinkedHashSet();
//En este ejemplo se crea una referencia a la interface Set
//pero se implementa con clases que usan la interface
```

Métodos default JDK8

Este modificador nos permite crear un nuevo método en la interfaz, pero definiendo su implementación por defecto, de tal manera que la clase que implementa la interface no requiera modificar los métodos con este modificador. Son métodos que tienen código en su interior.

Cuando usamos interfaces con métodos default, podemos encontrarnos con varias posibilidades a la hora de extenderlos:

• Que queramos dejar el método default sin cambios, y usarlo tal y como está.

• Podemos redeclarar el método default, convirtiéndolo en un método abstracto, que necesitaría implementación.
• Podemos redefinir el método default, lo cual lo sobreescribe y cambiaríamos el comportamiento de este.

Todo esto nos da múltiples posibilidades, podemos tener por ejemplo varias implementaciones de un método default extendiendo la interfaz inicial, y redefiniendo el método, para distintos comportamientos de las clases que implementen estos últimos. Podemos volverlo a convertir en abstracto, o podemos incluir comportamientos ya definidos en la interface padre. Y todas las combinaciones que se nos ocurran.

```
interface TestDefault{
      default void(){
            System.out.println(“Método default”);
      }
}
```

Los paquetes en Java

- Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.
- Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.
- Las clases tienen ciertos privilegios de acceso a los miembros dato y a las funciones miembro de otras clases dentro de un mismo paquete.

En el Entorno Integrado de Desarrollo (IDE), un proyecto nuevo se crea en un subdirectorio que tiene el nombre del proyecto. A continuación, se crea la aplicación, un archivo .java que contiene el código de una clase cuyo nombre es el mismo que el del archivo. Se pueden agregar nuevas clases al proyecto, todas ellas contenidas en archivos .java situadas en el mismo subdirectorio. La primera sentencia que encontramos en el código fuente de las distintas clases que forman el proyecto es **package** o del nombre del paquete.

```
//archivo MiApp.java
package nombrePaquete;
public class MiApp{
      //miembros dato
      //funciones miembro
}
```

La palabra reservada *import*

Para importar clases de un paquete se usa el comando **import**. Se puede importar una clase individual

```
import java.awt.Font;
```

o bien, se puede importar las clases declaradas públicas de un paquete completo, utilizando un arterisco (*) para reemplazar los nombres de clase individuales.

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

Para crear un objeto *fuente* de la clase *Font* podemos seguir dos alternativas

```
import java.awt.Font;
Font fuente=new Font("Monospaced", Font.BOLD, 36);
```

O bien, sin poner la sentencia **import**

```
java.awt.Font fuente=new java.awt.Font("Monospaced", Font.BOLD, 36);
```

Normalmente, usaremos la primera alternativa, ya que es la más económica en código, si tenemos que crear varias fuentes de texto.

Se pueden combinar ambas formas, por ejemplo, en la definición de la clase *BarTexto*

```
import java.awt.*;
public class BarTexto extends Panel implements java.io.Serializable{
      //...
}
```

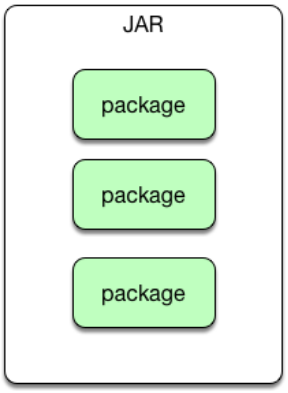
Panel es una clase que está en el paquete *java.awt*, y *Serializable* es un interface que está en el paquete *java.io*

Los paquetes estándar

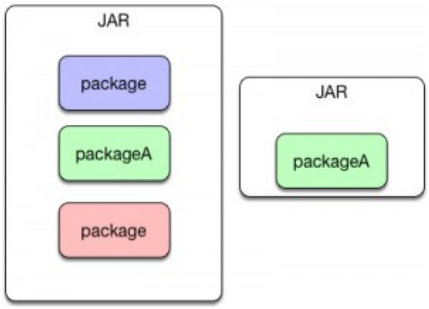
Paquete	**Descripción**
java.applet	Contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador
java.awt	Contiene clases para crear una aplicación GUI independiente de la plataforma
java.io	Entrada/Salida. Clases que definen distintos flujos de datos
java.lang	Contiene clases esenciales, se importa impícitamente sin necesidad de una sentencia **import**.
java.net	Se usa en combinación con las clases del paquete java.io para leer y escribir datos en la red.
java.util	Contiene otras clases útiles que ayudan al programador

Java 9 Modules y el concepto de modularidad

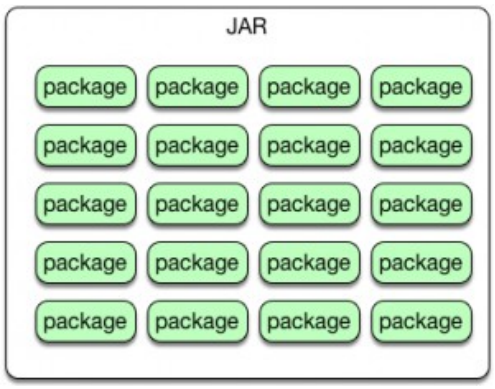
Hasta hoy en día Java ha organizado sus clases a través del concepto de paquetes que es un concepto puramente lógico. Un conjunto **de clases pertenecen a un paquete determinado**. Hasta aquí todo correcto . A nivel fisico varios packages son ubicados en un JAR o Java Archive.



Esto ha terminado siendo un poco pobre ya que es necesario **tener más de organización y modularidad** a la hora de trabajar **con grupos de clases y sus dependencias**. Por ejemplo clases de un mismo paquete podrían estar ubicadas en dos JARs diferentes.



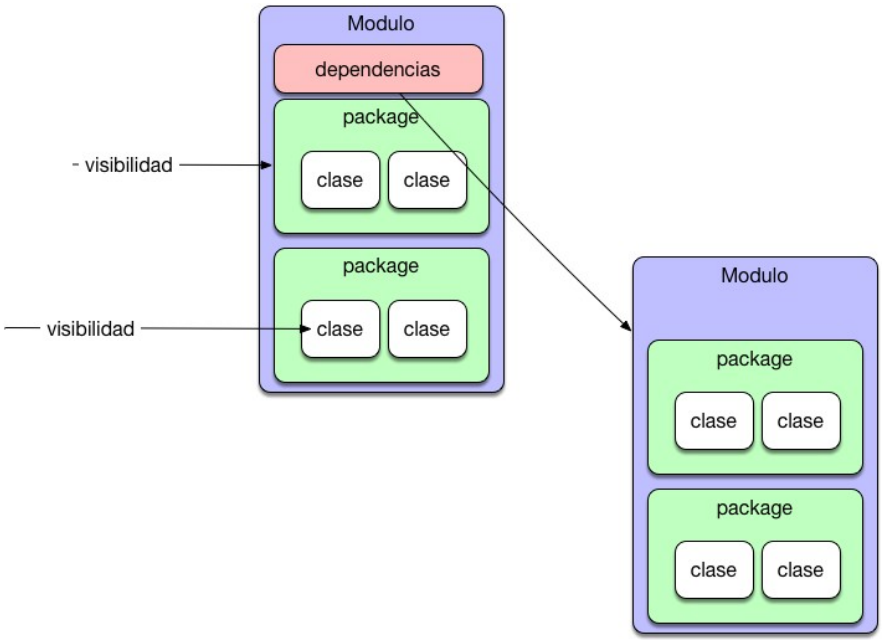
No solo eso sino que algunos de los ficheros JAR **a nivel de Java incluyen cientos de packages.** Por lo tanto estamos ante una situación que se acerca **bastante al concepto de monolito (una única pieza)**. Este es el caso mítico del **rt.jar** que agrupa a todas l**as clases core de Java y sus packages.**



Otro de los problemas que siempre han existido es como gestionar las dependencias entre un JAR y otro con los packages que están asociados. Maven siempre ha ayudado a ello , pero es cierto que es un **herramienta aparte** , no algo propio del lenguaje.

Java 9 Modules

Para solventar t**odos estos problemas Java 9 utiliza el concepto de módulo, algo que existe en otras plataformas como Node.** Un módulo es un conjunto de clases que pueden contener uno o varios packages y que define las dependencias con el resto de módulos **así como la visibilidad de las clases que contiene.**



Ejemplo de Java 9 Modules

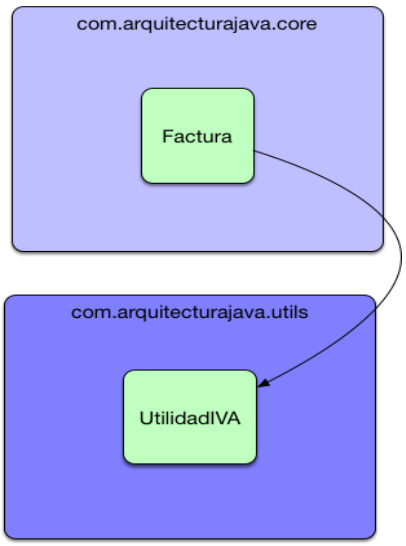
Vamos a construir un proyecto en el cual veamos **un ejemplo sencillo de los módulos**. Para ello nos vamos a construir un Utility Project. Recordemos que un proyecto de utilidades define una librería o JAR. En este proyecto vamos a incluir tres ficheros (Factura, Utilidades y module-info).

```
package com.arquitecturajava.core;
import com.arquitecturajava.utils.UtilidadIVA;

public class Factura {

	private int numero;
	private String concepto;
	private double importe;
	public int getNumero() {
		return numero;
	}
	public void setNumero(int numero) {
		this.numero = numero;
	}
	public String getConcepto() {
		return concepto;
```

```
	}
	public void setConcepto(String concepto) {
		this.concepto = concepto;
	}
	public double getImporte() {
		return importe;
	}
	public void setImporte(double importe) {
		this.importe = importe;
	}
	public double getImporteIVA() {
		return UtilidadIVA.calcularIVA(this.importe);
	}
}
```

```
package com.arquitecturajava.utils;
public class UtilidadIVA {

	public static double calcularIVA(double importe) {
		return importe *1.21;
	}
}
```

En este caso tenemos dos clases **Java relacionadas ubicadas en diferentes packages (core y utils)**.



Vamos a ver que información contiene el fichero que se encarga de la gestión de módulos. Si no existe en el proyecto lo creamos.

```
	module ModuloA {
		exports com.arquitecturajava.core;
	}
```

Es aquí donde podemos ver cual es la estructura de nuestro módulo.

Es un módulo que no tiene dependencias **pero que como peculiaridad no exporta todos los packages**. Únicamente se exporta el package **core que es el que contiene la clase Factura**. Es momento de usar nuestra librería en otro proyecto Java que tenga un fichero main.

```
package com.arquitecturajava;
import com.arquitecturajava.core.Factura;
import com.arquitecturajava.utils.UtilidadIVA; //error por que no hay
                        //acceso

public class Principal {

      public static void main(String[] args) {
            Factura f= new Factura();
            f.setImporte(200);
            System.out.println(f.getImporteIVA());
            System.out.println(UtilidadIVA.calcularIVA(0)); //error
      }
}
```

El código no funciona porque no podemos hacer uso de la clase **UtilidadIVA** ya que se encuentra **en un package que el módulo no publica**. Para poder utilizar el otro package hemos tenido que **añadir a nuestro proyecto main la dependencia del módulo.**

Java 9 Modules Export

Es ahora cuando podemos replantearnos si la encapsulación que hemos hecho de nuestros packages es la más correcta. En este caso es evidente que no. El calculo del IVA es una operación que puede ser usada por otras librerías o por el programa principal. Por lo tanto podemos modificar nuestro módulo y publicarla.

```
module ModuloA {
      exports com.arquitecturajava.core;
      exports com.arquitecturajava.utils;
}
```

Un módulo se define con las siguientes palabras clave:

`exports… to`	expone un paquete, opcionalmente a un módulo concreto
`import`	el típico import de Java. Lo normal es usar nombres completos de paquetes en vez de imports, pero si repites mucho un tipo, puede ser de utilidad.
`module`	Comienza la definición de un módulo.
`open`	Permite la reflexión en un módulo.

`opens`	Permite la reflexión en un paquete concreto, para alguno o todos los paquetes.
`provides… with`	Indica un servicio y su implementación.
`requires, static, transitive`	`requires` indica la dependencia con un módulo. Añade `static` para que sea requerido durante compilación y opcional durante la ejecución. Añade `transitive` para indicar dependencia con las dependencias del módulo requerido.

Clases anidadas

Cuando se crea una clase se puede declarar dentro de ella un atributo que sea referencia a un objeto. Esto permite implementar conceptos tan importantes como la agregación o la composición que definen distintos niveles de acoplamiento.
Sin embargo, existe otra posibilidad, la de declarar una clase dentro de otra. Java permite que se declare una clase dentro del bloque de declaración de otra y esto tiene consecuencias importantes respecto de las visibilidades que adquieren los elementos de ambas clases.
En primera instancia, parece que realizar esto no tiene mucho sentido porque la razón para hacerlo es que agrupa lógicamente clases que deben funcionar en conjunto (acoplamiento fuerte).
La pregunta es, ¿por qué realizar esto si supuestamente podría hacerse con una composición?
La respuesta a esta pregunta radica en las capacidades declarativas de una clase y su fundamentación reside en las visibilidades, las capacidades de herencia única y la codificación asociadas a éstas. Por ejemplo, una clase, por más que se declare dentro de otra, tiene la capacidad de heredar e implementar tantas interfaces como quiera.
Quedan por resolver las cuestiones de visibilidades de sus miembros. Por supuesto que se siguen respetando las reglas de bloques definidas con anterioridad, pero al utilizarlas en las invocaciones a elementos de la clase que anida a otra o creaciones de objetos del tipo de la clase que se encuentra anidada dentro de otra, se debe tener cuidado de tener bien definida la visibilidad para acceder.
Los miembros de la clase anidada tienen acceso a los miembros de la clase que la anida, por lo tanto, esta regla ya expuesta para los bloques de sentencia se respeta, sin embargo, se agregan nuevas a tener en cuenta porque la definición ahora incluye a “toda” una clase.

Propiedades de las clases anidadas

Las reglas para trabajar con clases anidadas están basadas en las propiedades que se definen a continuación:

- El nombre de la clase anidada deberá ser diferente al de la que la anida.
- Se puede utilizar el nombre de la clase en una declaración sólo dentro del bloque en el que está definida. Para usarla fuera de él se debe resolver visibilidad con un nombre calificado.
- Pueden declararse inclusive dentro de un método.
- Una clase anidada se puede declarar como public , por defecto, private , protected , abstract , final o static .
- Una clase anidada sólo puede tener un elemento static sólo si toda la clase es declarada con el mismo modificador.

```
public class Exterior1 {
	private int var;
	/* Declaración de una clase anidada llamada Anidada */
	public class Anidada {
		public void haceAlgo() {
			// La clase anidada tiene acceso a la variable var
			// de la clase exterior
			var++;
		}
	}
	public void verificaAnidada() {
		Anidada i = new Anidada();
		i.haceAlgo();
	}
}
```

Un ejemplo un poco más extremo es el de declarar una clase anidada dentro de un método de la clase que la anida. El problema de este caso es que la visibilidad de la clase estará seriamente limitada a la visibilidad del método dentro del cual se la define.

```
public class Exterior3 {
	private int var = 5;
	public Object instanciaAnidada(int varLocal) {
		final int varLocalFinal = 6;
		// Declaración de una clase dentro de un método
		class Anidada {
			public String toString() {
				return ("#<Anidada var=" + var +
				// " varLocal =" + varLocal + // ERROR: ILEGAL
				" varLocalFinal=" + varLocalFinal + ">"); // CONSTANTE
			}
		}
		return new Anidada();
	}
	public static void main(String[] args) {
		Exterior3 afuera = new Exterior3();
		Object obj = afuera.instanciaAnidada(47);
		System. out .println("El objeto es " + obj);
	}
}
```

En este ejemplo se puede observar que:

- La variable varLocal (parámetro del método) no es accesible para los miembros de la clase. Esto sucede porque sólo puede resolverse la visibilidad de los elementos fuera de su bloque de declaraciones con this , pero las variables locales o parámetros no tiene un this asociado ya que no pertenecen a la clase y si al método que las define.
- Se puede acceder a varLocalFinal porque por ser una constante el compilador la optimiza remplazando su declaración por el valor. Como consecuencia, en tiempo de ejecución no hay que resolver ninguna referencia porque en ese lugar ahora se encuentra una constante.
- Para crear un objeto del tipo de esta clase se la “debe poder ver”, pero esto sucede sólo dentro de la visibilidad del método, por lo tanto, sólo ahí se puede crear una instancia de ella. Observar que en el ejemplo esto se realiza retornando una referencia a un nuevo objeto que se crea “en el método”.
- Declaraciones de este tipo son muy útiles cuando se quieren implementar métodos que se encarguen exclusivamente de la creación de objetos y que sólo ellos puedan “fabricarlos”.

Clases anónimas

Existen situaciones en las cuales se crea un objeto como un hecho puntual. No se desea que esos objetos estén registrados para otras invocaciones sino que se los quiere crear sólo cuando son necesarios. Tienen las siguientes propiedades:

- Son clases que crean objetos sólo con su bloque de declaración, por lo tanto no se puede crear más de una referencia de su tipo por objeto.
- Siempre están acompañadas de un operador new porque sólo se puede crear una referencia cuando se las declara.
- Pueden estar declaradas en cualquier parte del código siempre y cuando se asocien a un elemento de código que pueda almacenar la referencia que se devuelve al crear el objeto.

Uno de los problemas que soluciona es, como se mencionó anteriormente, la creación de un objeto dentro de otro, donde la condición es que existe una referencia única del objeto del tipo de la clase anidada en objeto que la anida.
Estas clases son como las clases anidadas salvo que no tienen un nombre que las identifique en su declaración.

```
public interface Saludo {
	public static final String DEFAULT = "Adiós";
	public void imprimirSaludo();
}

public class Anonima {
	public static void main(String[] args) {
		// Creación de un objeto anónimo
		Saludo sAnonimo = new Saludo() {
			public void imprimirSaludo() {
				System. out .println("Saludo: " + DEFAULT );
			}
		};
		Anonima a = new Anonima();
		a.saludar(sAnonimo);
		// Creación de un objeto anónimo en el momento que se
		// pasa el argumento
		a.saludar(new Saludo() {
			public void imprimirSaludo() {
				System. out .println("Saludo desde otro objeto: " +
				DEFAULT );
			}
		});
	}
	public void saludar(Saludo s) {
		s.imprimirSaludo();
	}
}
```

Clases anidadas y anónimas

También se puede utilizar clases anónimas creándolas dentro de métodos, como se hizo anteriormente con las clases anidadas. La unión de estos ejemplos respeta todo lo enunciado en los casos que se analizó por separado. La diferencia fundamental es que la referencia se guarda primero en una variable llamada obj antes que el método la retorne.

```
public class Exterior {
	private int var = 5;
	public Object instanciaAnidadaAnonima(int varLocal) {
```

```
            final int varLocalFinal = 6;
            // Declaración de una clase anónima dentro de un método
            Object obj = new Object() {
                  public String toString() {
                        return ("#<Anidada var=" + var +
                        // " varLocal=" + varLocal + // ERROR: ILEGAL
                        " varLocalFinal=" + varLocalFinal + ">");
                  }
            };
            return obj;
      }


      public static void main(String[] args) {
            Exterior afuera = new Exterior();
            Object obj = afuera.instanciaAnidadaAnonima(47);
            System. out .println("El Objeto es " + obj);
      }
}
```

Enumeraciones

Una enumeración es una construcción que permite el lenguaje para definir constantes de tipo seguro, esto, constantes con el tipo asegurado durante la ejecución de un programa. Las enumeraciones pueden tener una o varias constantes como sus elementos. Todas las enumeraciones derivan implícitamente de java.lang.Enum y definen instancias únicas de una enumeración específica. Por lo tanto, cada enumeración define un tipo al igual que una clase y se la puede tratar de forma similar, ya que permiten definir sus elementos como si fueran variables de instancia.

El manejo de las constantes

Antes de la versión 1.5, el manejo de constantes estaba delimitado a las declaraciones de tipo final . Así si se planteaba un problema en el cual se querían manejar una serie de constantes era común crear una solución con variables estáticas públicas finales para poder accederlas de forma simple. Por ejemplo, si se tiene como problema determinar una serie de niveles de seguridad constantes a los cuales se les asigna una serie de grados según el nivel, se diseñaba la clase definiendo los niveles constantes dentro de ella.

El nuevo tipo enumerado

A partir de la versión 5.0 de Java SE se incluye una modalidad de tipos enumerados que mantiene la seguridad de estos (type safe). En el código se muestra un tipo enumerado para representar los niveles de seguridad. Se debe pensar el “tipo” del nivel de seguridad como en una clase con un conjunto finito de valores que reciben los nombres simbólicos incluidos en la definición de dicho tipo.

```
public enum Nivel {
      INICIAL,USUARIO, MANEJADOR_DE_RECURSOS, ADMINISTRADOR
}
```

Al usar un tipo enumerado, se sabe exactamente los valores que pueden asumir las constantes declaradas de ese tipo, con lo cual no hay que verificar que asuma algún valor diferente. Estos valores se los considera valores estáticos porque tienen las mismas propiedades de acceso que una constante declarada de esa manera. Esto tiene un impacto directo sobre el código de la clase cliente de la enumeración, porque los tipos a partir de su utilización se encuentran asegurados.

Tipos enumerados avanzados

Refinamiento de la enumeración para incluir nombres de constantes.
Obsérvese cómo se oculta la información de forma adecuada con modificador de visibilidad private y el método de acceso se define como public .

```
public enum Nivel {
      INICIAL("Inicial"),USUARIO("Usuario"),
      MANEJADOR_DE_RECURSOS("Manejador de Recursos"),
      ADMINISTRADOR("Administrador");
      private final String nombre;
      private Nivel(String nombre) {
            this.nombre = nombre;
      }
      public String getNombre() { return nombre; }
}
```

Un constructor enum siempre debería utilizar un modificador de acceso privado. Los argumentos del constructor se suministran después de cada valor declarado. Por ejemplo, la línea que define la cadena "Inicial" es el argumento del constructor del enum para el valor INICIAL . Los tipos enumerados pueden tener cualquier cantidad de atributos y métodos.

Importaciones estáticas

Cuando una enumeración no se encuentra en el mismo paquete que la clase que la utiliza, es necesario importarla.
Sin embargo, si se necesita acceder a los miembros de la enumeración, será necesario calificar las referencias con ésta. Esto se aplica también a los valores de los tipos de enumeración, como se muestra en los casos de Nivel.ADMINISTRADOR o Nivel.USUARIO .
A partir de la versión 5.0 de J2SE se proporciona la funcionalidad de la importación estática que permite acceder a los miembros estáticos (recordar que se consideran de esta manera a los elementos de una enumeración) sin tener que calificarlos con el nombre de la clase.

```
      Nivel nivel=Nivel.ADMINISTRADOR;    //referencia calificada
      Nivel nivel=ADMINISTRADOR;          //referencia sin calificar
```

Atención: Procurar ser cauto al usar estas importaciones. Si abusa de ellas, puede hacer que el programa se convierta en un código ilegible y difícil de mantener ya que contaminará su espacio de nombres con todos los miembros estáticos que importe. Las personas que lean el código en el futuro no sabrán de qué clase procede cada miembro estático. La importación de todos los miembros estáticos de una clase puede afectar muy negativamente a la legibilidad. Si sólo necesita uno o dos miembros, importarlos de forma individual. Si se usa con prudencia, la importación estática puede facilitar la lectura del código, ya que elimina la repetición estándar de los nombres de enumeraciones.