Unidad

3

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 5

Clases: Conceptos Avanzados

Clases: Conceptos Avanzados

En este Capítulo

- La palabra reservada static
- La Clase Object
- La Palabra Clave final
- Clases Abstractas
- Polimorfismo
- Interfaces
- Clases Anidadas
- Clases Anónimas

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 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.

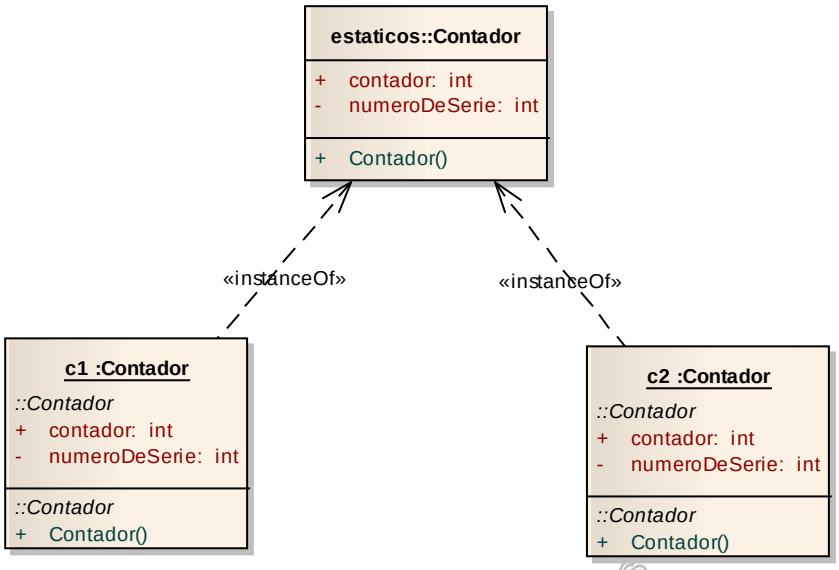
Como no se necesita un objeto del tipo de la clase, indica claramente que estos elementos no pertenecerán al objeto que se crea a partir de la clase que los contiene y por lo tanto no podrán acceder tampoco a los miembros de un objeto simplemente "porque no los ve". La razón de fondo para esto es que 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 (nombreDeClase.nombreDeAtributo) podrá ser accedido por cualquier clase.

A continuación, y como ejemplo, se muestra el diagrama de una clase en UML junto con dos objetos, c1 y c2, que son instancias del tipo Contador. Notar que el estereotipo del vínculo en el diagrama (instanceOf) indica que son dos objetos del tipo Contador. Además, c1 y c2 son diagramas de objetos mientras que Contador es de clase. Notar también que, en esta última, aparece el atributo contador subrayado y en UML esto indica que el atributo es estático



La forma de acceso a un atributo estático desde otra clase se muestra en el siguiente código.

Ejemplo

```
package estaticos;
public class Contador {
	public static int contador = 0;
	private int numeroDeSerie;

	public Contador() {
		contador++;
		numeroDeSerie = contador;
	}
}


package estaticos;
public class OtraClase {
	public void incrementarNumero() {
		Contador.contador++;
	}
}
```


Notar que al estar el atributo contador en el constructor y al ser estático, cada vez que se cree un objeto ocasionará que se incremente el valor almacenado en esa área de memoria. Como todos los objetos de tipo Contador comparten esta área de memoria, cada vez que se cree un objeto se incrementará sobre valor al ejecutar el constructor, lo cual deviene “en una especie” de contador de objetos del mismo tipo.

Métodos static

Se puede invocar un método **static** sin una instancia de la clase a la que pertenece. 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 y esto implica que se deberá resolver la visibilidad por medio de un nombre totalmente calificado con notación de punto para accederlo.

Ejemplo

```
package estaticos.conthis;
public class Contador {
	public static int contador = 0;
	private int numeroDeSerie;

	public Contador() {
		contador++;
		numeroDeSerie = contador;
	}

	public static int getContador() {
		return contador;
	}
}
```

Por lo tanto, si se quiere acceder el método, se deberá resolver la visibilidad de acceso al mismo con un nombre totalmente calificado que incluya al de la clase. El siguiente ejemplo muestra esta situación.

Ejemplo

```
package estaticos.conthis;

public class VerificarContador {
	public static void main(String[] args) {
		System.out.println("El valor de contador es " +
			Contador.getContador());
		Contador c = new Contador();
		System.out.println("El valor de contador es " +
			Contador.getContador());
	}
}
```

La salida del programa es:

```
El valor de contador es 0
El valor de contador es 1
```

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 invocarlos al comienzo de cada programa que los incluya.

Ejemplo

```
package estaticos.singleton;
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 devolver 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

Un ejemplo de uso se presenta a continuación:

```
package estaticos.singleton;
public class VerificarInstanciaUnica {
	public static void main(String[] args) {
		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

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

Ejemplo

```
package finales;
public class Pi {
	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 inicia una variable estática.

Ejemplo

```
package finales;

public class Constantes {
	public final int VALOR_MAXIMO = 10;
	public final int VALOR_INICIAL;

	public Constantes(int i) {
		VALOR_INICIAL = i;
	}
}
```

Métodos

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. Los métodos **static** o **private** son final automáticamente

Ejemplo

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

package finales;
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

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 y tiene un formato como el siguiente:

```
package finales;

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.

Aunque no se incluya en la declaración de una clase, implícitamente se deriva como si estuviese declarado.

Ejemplo

```
package object;
public class Empleado{

}
```

Es lo mismo que:

```
package object;
public class Empleado extends Object{

}
```

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.

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. La implementación por defecto, la que está escrita como código en Object, sólo compara la igualdad de ambas referencias.

```
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.

```
public native int hashCode();
```

Una implementación basada en la igualdad que definen equals y hashCode es un valor por defecto sensible, pero para algunas clases, es conveniente delegar la definición de igualdad a otro elemento que la defina. Por ejemplo, la clase Integer define 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.

Sin embargo, se debe cumplir con ciertos requisitos para que los programas funcionen correctamente a las que se denominan relaciones de equivalencia. Éstas se presentan en la siguiente tabla:

Relaciones de Equivalencia de equals	
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**

A continuación, se presenta la comparación de objetos en una clase utilizada en el módulo anterior a la que se le sobrescribe el método equals():

Ejemplo

```
package object.equals;
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;
		temp = Double.doubleToLongBits(salario);
		result = prime * result + (int) (temp ^ (temp >>> 32));
		return result;
	}

	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;
      }

}
```

Se puede comprobar el uso de las relaciones de equivalencia en el siguiente gráfico

```
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;
}
```

Reflexividad

No nulificable

Asegurar el tipo para convertir la referencia a su tipo original - **Simetría**

Comparar otros objetos usando su propia definición de equals()

Convertir a Bits la comparación para evitar errores

Transitividad

Consistencia

Para probar el ejemplo anterior, se implementa una clase que cree dos objetos del mismo tipo y los compare, tanto en sus referencias como en sus atributos:

Ejemplo

```
package object.equals;
public class VerificaEmpleado {

      public static void main(String[] args) {
            Fecha f1 = new Fecha(10, 10, 2012);
            Empleado e1 = new Empleado("Juan", f1);
            Empleado e2 = new Empleado("Juan", f1);

            System.out.println(e1.equals(e2));
            System.out.println(e1.equals(new Empleado("Juan")));

            if ( e1 == e2 ) {
```

```
                System.out.println("e1 es identico a e2");
            } else {
                System.out.println("e1 no es identico a e2");
            }
            if ( e1.equals(e2) ) {
                System.out.println("e1 es igual a e2");
            } else {
                System.out.println("e1 no es igual a e2");
            }
            System.out.println("asignar e2 = e1;");
            e2 = e1;
            if ( e1 == e2 ) {
            System.out.println("e1 es identico a e2");
            } else {
                System.out.println("e1 no es identico a e2");
            }
        }
}
```

El programa que prueba la comparación de objetos presentado anteriormente tiene condicionada su salida por una serie de bifurcaciones condicionales.

La salida que se obtiene es:

```
true
false
e1 no es identico a e2
e1 es igual a e2
asignar e2 = e1;
e1 es identico a e2
```

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 Strings 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.

Ejemplo

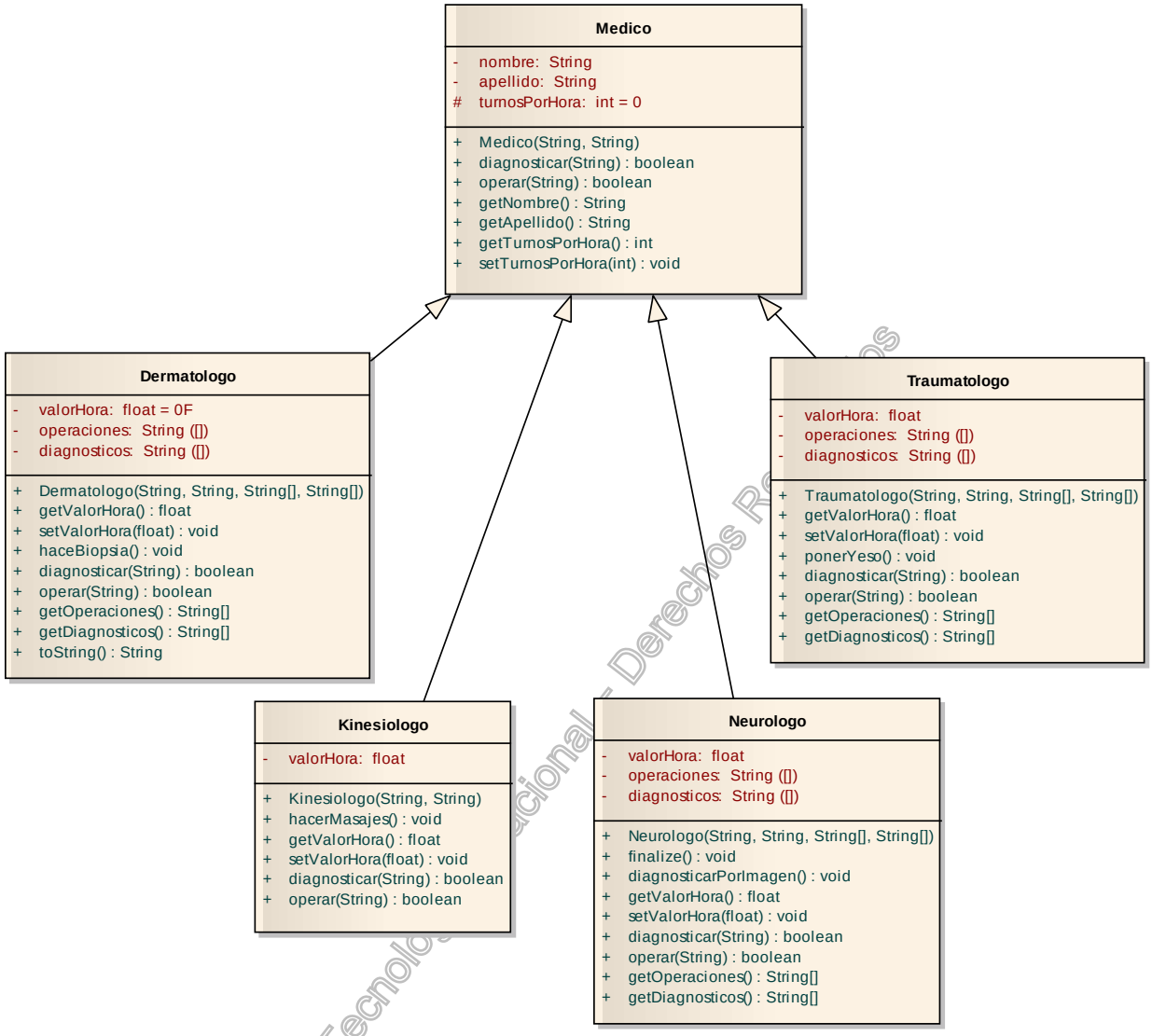
```
public String toString() {
	return "Empleado [nombre=" + nombre + ", salario=" + salario
			+ ", fechaNacimiento=" + fechaNacimiento + "]";
}
```

Clases Abstractas

Escenario inicial:

Suponiendo que un sistema necesita generar una clase Medico para modelar los médicos que pertenecen a una clínica. Se sabe que la empresa necesita manejar distintas especialidades en las cuales los médicos prestan servicios como realizar diagnósticos u operaciones (aunque hay algunos que no realizan este servicio).

El diagrama UML que describe la interacción de las clases es el siguiente:



Código inicial

La clase Clinica es la encargada del uso de los objetos del tipo Medico iniciando el valor de su posible creación.

Ejemplo

```
public class Clinica {
	public static void main(String[] args) {
		AdministracionDeTurnos adt =
AdministracionDeTurnos.getAdministracionDeTurnos();
		String operacionest[] = {"neurofibromatosis", "hernia de disco",
			"fracturas simples", "fracturas expuestas"};
		String diagnosticost[] = {"lumbalgia", "psoriasis", "fracturas",
			"dolores", "cervicales"};
```

```
            Traumatologo t = new Traumatologo("Iván", "Lacarra", operacionest,
                  diagnosticost);
            adt.agregarMedico(t);

            String operacionesd[] = {"neurofibromatosis","carcinoma",
                  "lipoma"};
            String diagnosticosd[] = {"liquen plano", "psoriasis", "lepra"};
            Dermatologo d = new Dermatologo("Pedro", "Rivero", operacionesd,
                  diagnosticosd);
            adt.agregarMedico(d);

            Kinesiologo k = new Kinesiologo("Alberto", "Tati");
            adt.agregarMedico(k);

            String operacionesn[] = {"neurofibromatosis", "neurinoma"};
            String diagnosticosn[] = {"vértigo", "psoriasis", "náuseas",
                  "ACV"};
            Neurologo n = new Neurologo("Pedro", "Pomar", operacionesn,
                  diagnosticosn);
            adt.agregarMedico(n);

            ReporteDeMedicos rm = new ReporteDeMedicos();
            rm.generarReporteActividades();
      }
}
```

Código de ReporteDeMedicos

La clase encargada de generar los reportes necesita acceder a la clase de instancia única que tiene a su cargo la administración de turnos y doctores de la clínica.

Ejemplo

```
public class ReporteDeMedicos {

      public void generarReporteActividades() {
            AdministracionDeTurnos adt = AdministracionDeTurnos
                        .getAdministracionDeTurnos();
            Medico listaMedicos[] = adt.getListaMedicos();
            int indice = adt.getIndice();
            String tipoDiagnostico = "psoriasis";
            String tipoOperacion = "neurofibromatosis";

            for (int i = 0; i < indice; i++) {
                  System.out.println("El médico " + listaMedicos[i].getNombre()
                        + " " + listaMedicos[i].getApellido());
                  if (listaMedicos[i].operar(tipoOperacion))
                        System.out.println("Opera " + tipoOperacion);
                  else
                        System.out.println("No opera");
                  if (listaMedicos[i].diagnosticar(tipoDiagnostico))
                        System.out.println("Diagnostica " + tipoDiagnostico);
                  else
                        System.out.println("No diagnostica");
```

```
            }
        }
}
```

El problema de este código radica en que sólo las subclases tendrán “el conocimiento” de cómo calcular tanto los diagnósticos como las operaciones que realizan los médicos. Con las técnicas explicadas hasta el momento, se debe crear un método con cuerpo vacío para que las subclases lo puedan rescribir.

Definiciones

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 completa, incluso proporciona a sus subclases la declaración de todos los métodos necesarios para implementar la 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 declararlos todos a partir de un mismo objeto padre, 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 ObjetosGraficos deben saber como 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). La clase ObjetoGrafico se parecería a esto:

```
abstract class ObjetoGrafico {
        int x, y;
        . . .
        void moverA(int nuevaX, int nuevaY) {
        . . .
        }
        abstract void dibujar();
        }
}
```

Todas las subclases no abstractas de ObjetoGrafico como son Circulo o Rectangulo deberán proporcionar una implementación para el método dibujar().

El código a continuación muestra esta situación

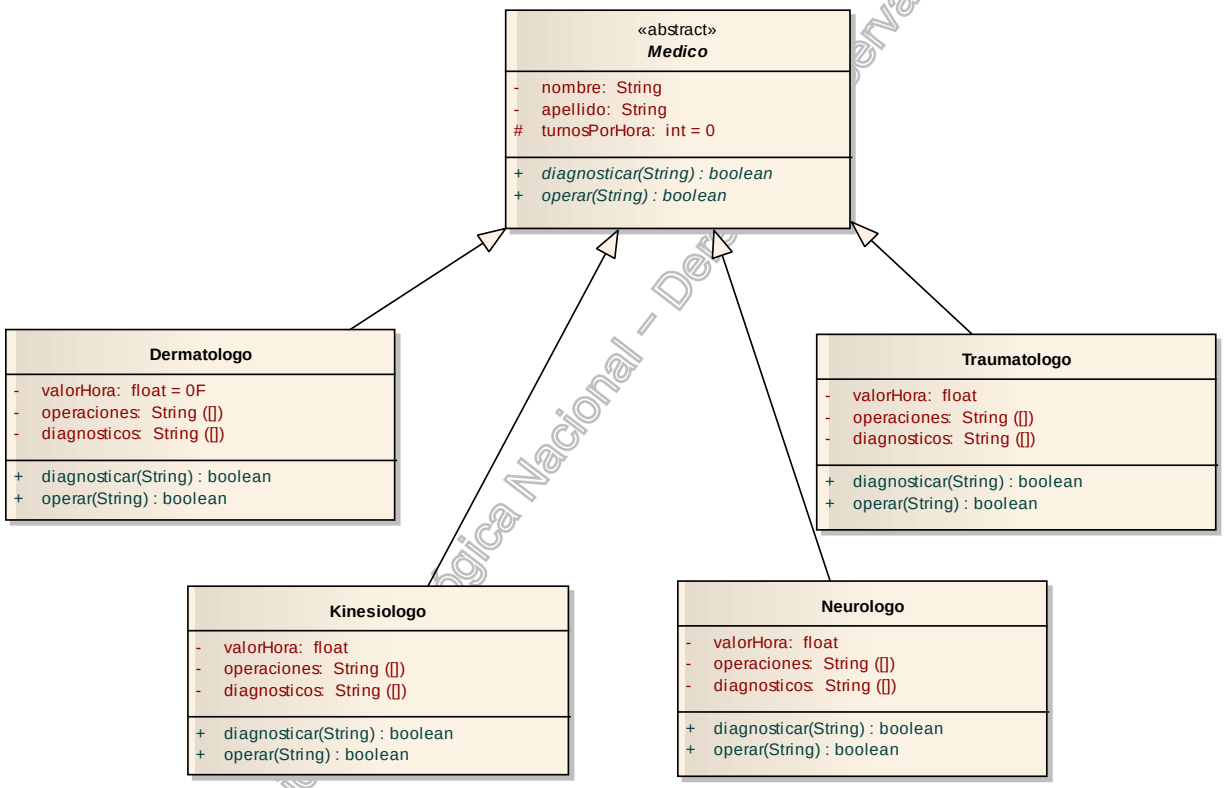
```
class Circulo extends ObjetoGrafico {
        void dibujar() {
        . . .
        }
}



class Rectangulo extends ObjetoGrafico {
        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 sus superclases debe ser declarada como una clase abstracta, o 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

Un ejemplo de solución al problema anterior de los vehículos se muestra en el gráfico

Ejemplo



Código

```
package abstractas.fin;
public abstract class Medico {
	private String nombre;
	private String apellido;
	protected int turnosPorHora = 0;

	public Medico(String nombre, String apellido) {
		this.nombre = nombre;
		this.apellido = apellido;
	}
```

```
	public abstract boolean diagnosticar(String tipoDiagnostico);

	public abstract boolean operar(String tipoOperacion);

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getApellido() {
		return apellido;
	}

	public void setApellido(String apellido) {
		this.apellido = apellido;
	}

	public int getTurnosPorHora() {
		return turnosPorHora;
	}

	public void setTurnosPorHora(int turnosPorHora) {
		this.turnosPorHora = turnosPorHora;
	}
}
```

Código de uno de los médicos posibles

```
package abstractas.fin;
public class Dermatologo extends Medico {

	private float valorHora = 0F;
	private String operaciones[];
	private String diagnosticos[];

	public Dermatologo(String nombre, String apellido, String operaciones[],
			String diagnosticos[]) {
		super(nombre, apellido);
		this.operaciones = operaciones;
		this.diagnosticos = diagnosticos;
	}

	public float getValorHora() {
		return valorHora;
	}

	public void setValorHora(float valorHora) {
		this.valorHora = valorHora;
	}

	public void haceBiopsia() {
```

```
	}

	public boolean diagnosticar(String tipoDiagnostico) {
		for (int i = 0; i < diagnosticos.length; i++) {
			if (diagnosticos[i] != null
					&& diagnosticos[i].equals(tipoDiagnostico))
				return true;
		}
		return false;
	}

	public boolean operar(String tipoOperacion) {
		for (int i = 0; i < operaciones.length; i++) {
			if (operaciones[i] != null &&
				operaciones[i].equals(tipoOperacion))
				return true;
		}
		return false;
	}
}
```

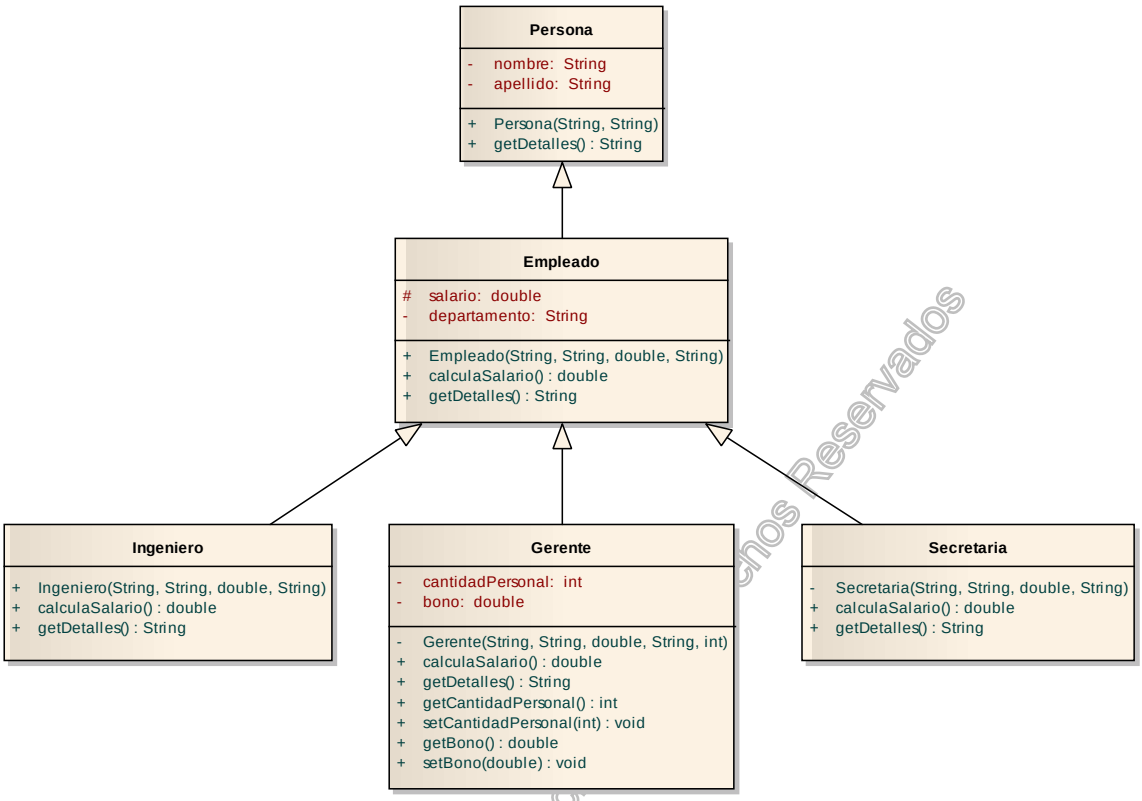
Polimorfismo

El polimorfismo se manifiesta cuando se conoce la operación a realizar pero se desconoce cuando 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.

La forma más sencilla de ver esto es a través de un ejemplo como el que se muestra en el siguiente diagrama UML que plantea una serie de herencias (tres para ser más exactos) y métodos en común en las distintas clases que las componen.

En el ejemplo existen dos métodos que se sobrescriben, uno, getDetalles, desde la superclase base de todas las cadenas de herencias, Persona, y otro desde la clase Empleado, calculaSalario. En base a la reescritura de estos métodos y el ocultamiento de la visibilidad que se produce por ella, se plantearán los conceptos base del polimorfismo.

Ejemplo



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

Los objetos tienen una sola forma, pero las acciones que se ejecuten en ellos pueden ser específicas de cada uno. De esta manera la acción getDetalles() se llamará igual en las clases Persona, Empleado y Gerente, pero el resultado como la acción en si misma será diferente. Lo mismo ocurre en el caso del método calculaSalario. Notar que este último se empieza a implementar a partir de la clase Empleado, por lo tanto, un método no debe estar en la superclase de la cadena de herencia para poder implementar polimorfismo

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 esta 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 este definido en la superclase será un error en tiempo de compilación.

Ejemplo

```
package empleados;

public class VerificaPersonal {

	public static void main(String[] args) {
		VerificaPersonal vp = new VerificaPersonal();
		Empleado e = new Gerente("Pedro", "Gonzalez", 100000, "Sistemas",
					15); //legal
		e.setCantidadPersonal(15); // Intento ilegal de acceder a
		//un atributo del tipo Gerente
		// La variable de referencia es declarada como un tipo
		// Empleado, por lo tanto puede acceder a elementos que
		// existan en Empleado, aunque el objeto sea de tipo Gerente
		:
		:
	}
}
```

Cuando las asignaciones están correctas, se pueden hacer invocaciones virtuales (referencias polimórficos).

Ejemplo

```
Empleado e = new Gerente("Pedro", "Gonzalez", 100000, "Sistemas", 15); //legal
		:
		:
e.getDetalles();
		:
		:
```

Para valerse de la oportunidad que brinda el ocultar un método cuando está rescrito, se deben crear objetos de las clases que se utilizarán. Por ejemplo, dentro de la clase empleado se puede ver uno de los métodos rescritos.

Ejemplo

```
package personal;



public class Empleado extends Persona {

		:
		:

	public double calculaSalario() {
		return salario = salario - salario * 0.21;
	}

		:
		:
}
```

También se puede observar el mismo método dentro de la clase Gerente

```
public class Gerente extends Empleado {

		:
		:

	public double calculaSalario() {
		return salario = salario - salario * 0.21 + bono;
	}

		:
		:
}
```

Para utilizar la capacidad de invocar métodos virtuales, una tercera clase crea objetos de Empleado y Gerente para valerse del ocultamiento de métodos por rescritura al momento de realizar la invocación. Para ello utiliza la superclase de la sub cadena de herencia a partir de la clase Empleado y declara una referencia de este tipo para asignarle valores de referencias de objetos de su tipo o de los de una subclase de ella.

Ejemplo

```
package empleados;
public class VerificaPersonal {

	public static void main(String[] args) {
		VerificaPersonal vp = new VerificaPersonal();
		:
		:
		Gerente g = new Gerente("Juan", "Perez", 10000, "Sistemas", 15);
		Empleado e = new Empleado("Sebastián", "Dominguez", 5000,
					"Sistemas");
		double t = vp.pol(g);
```

```
            t = vp.pol(e);
      }

      public double pol(Empleado e) {
            return e.calculaSalario();
      }
}
```

Interfaces

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.

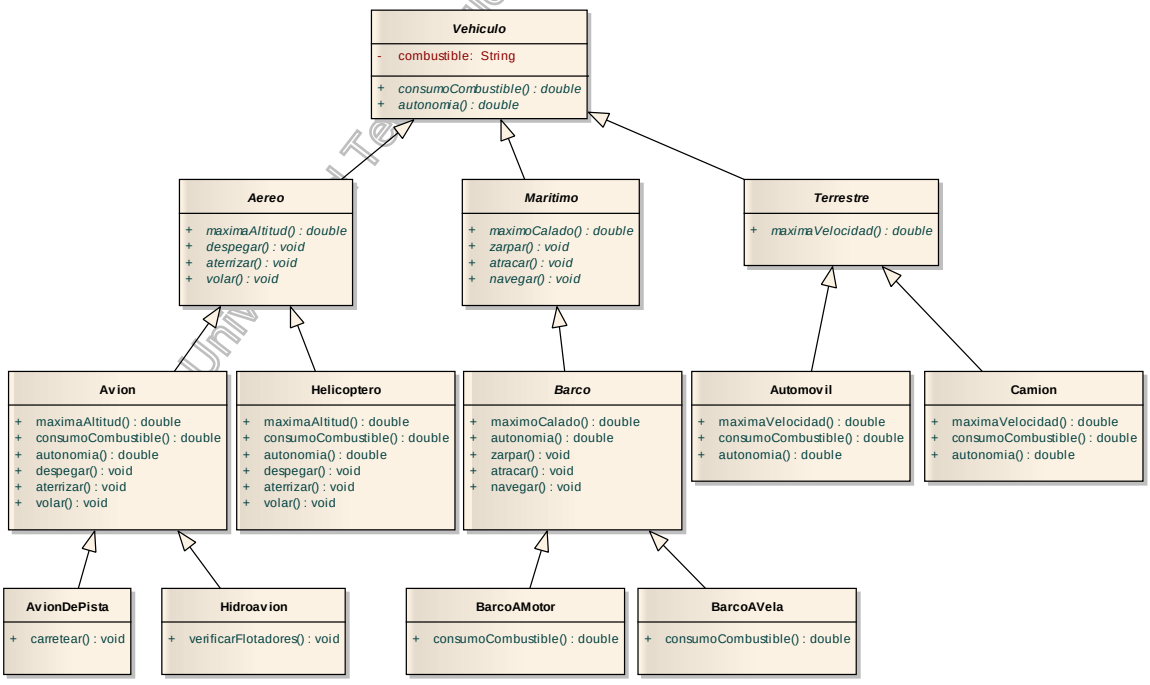
Capturan similitudes entre clases no relacionadas sin forzar relaciones de asociación, agregación, composición o herencia entre ellas, o, mejor dicho, cuando dos clases realizan las mismas acciones pero pertenecen a cadenas de herencia diferente, pueden compartir invocaciones virtuales de métodos si 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

Ejemplo

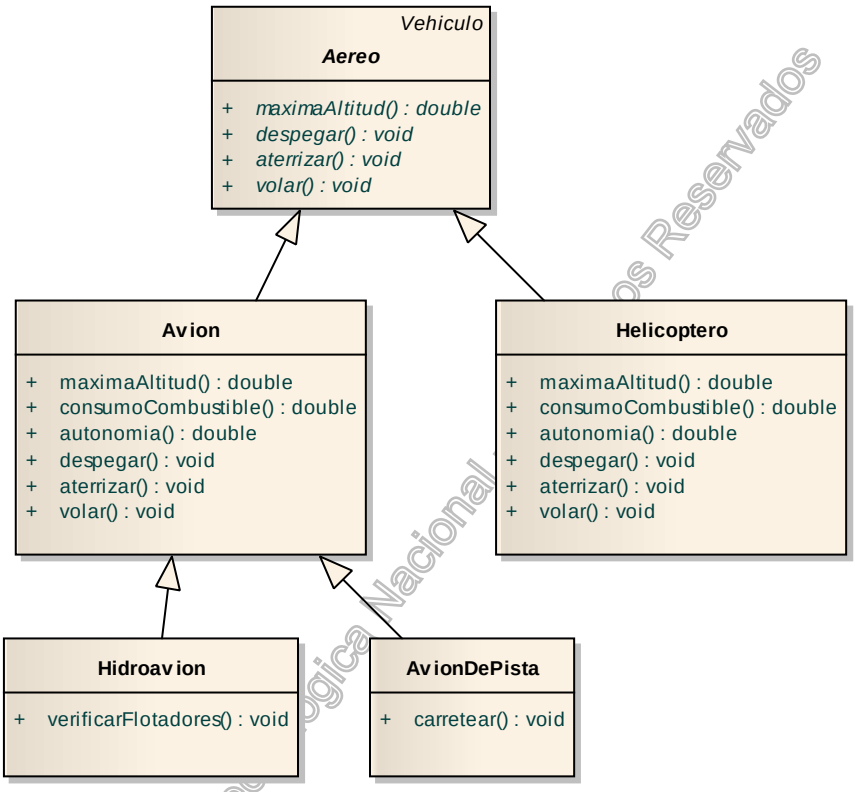
De la misma forma que con el polimorfismo, es más fácil comprender las posibilidades brindadas por las interfaces a través de un ejemplo. Como punto de partida se supone un diseño de herencias de clases como muestra el gráfico UML. Recordar que este tipo de gráficos no necesita exponer todos los miembros de clases, sólo los más importantes.

Ejemplo



Dentro del contexto del ejemplo, sólo interesa analizar una cadena de herencia, en la cuál se pueden observar los métodos que se implementan en la clase Aéreo. Si se observa bien esta clase, se puede apreciar que posee algunos métodos abstractos y que la clase en si misma también lo es. Esto es lógico si tenemos en cuenta que la información de cómo realizar estas acciones estará disponible en las subclases.

Ejemplo



Refinamiento

Cuando se analizan con detalles este tipo de clases se puede detectar métodos que reflejan acciones que se pueden llevar a cabo por más de un tipo de objeto y, además, muchos de ellos ni siquiera están en la misma cadena de herencia. Este hecho se puede detectar en la etapa de análisis, de diseño o, inclusive, en una etapa de desarrollo cuando se compara con otras cadenas de herencia.

Cuando las clases definen objetos de distintos tipo que realizan las mismas acciones se descubren comportamientos similares entre objetos. Estos se pueden relacionar a través de una referencia en común donde se declaren dichas acciones.

Con esta referencia, se puede llamar a esos métodos cuando los objetos (del mismo tipo u otro en donde están declarados los métodos) deben brindar un servicio que necesita dichas operaciones para llevarse a cabo.

Para crear esa referencia en común, se extraen en un refinamiento los métodos que tengan estas características y se los coloca en una interfaz, para que luego cada clase la implemente y rescriba dichos métodos.

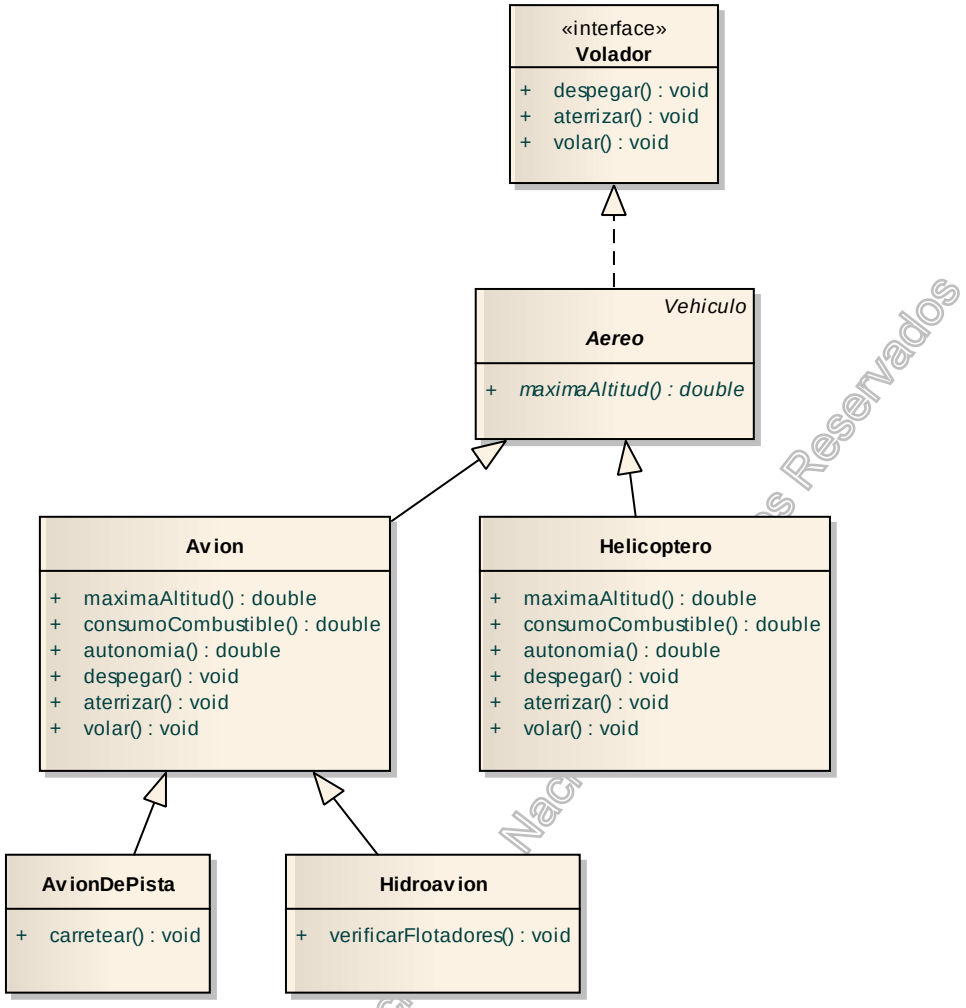
Otro detalle muy importante a tener en cuenta es que no sólo se extraen métodos al azar, sino aquellos que en conjunto puedan definir un concepto por si mismos (muchas veces esto sucede cuando un grupo de servicios son necesarios pero no lo suficientemente significativos en el contexto de un sistema para conformar un objeto por si mismo). En el ejemplo que se muestra a continuación, los métodos que se extraen son características inherentes de aquellos que son voladores, por eso se define la interfaz con el nombre Volador.

Es claro que el lugar que se implemente la interfaz no debe necesariamente ser donde se rescriban los métodos que esta define (recordar que los métodos de una interfaz son abstractos y deben rescribirse obligatoriamente para crear objetos de dicha clase), sino al menos, en una clase que la anteceda en la cadena de herencia.

En el ejemplo que se muestra a continuación, y sólo a modo de claridad en la exposición, se presenta la implementación de la interfaz al mismo nivel en el cuál se descubren los métodos con los que se compondrá la interfaz, pero esta claro que las clases que los rescribirán para convertirlos en métodos concretos, son sus subclases.

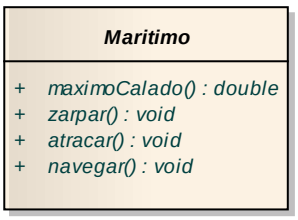
Sin embargo, en el caso de la clase Aéreo esto no afecta por ser abstracta, por lo tanto si no se sobrescriben los métodos no afecta su comportamiento porque antes de este refinamiento tampoco se podían crear objetos de este tipo.

Ejemplo



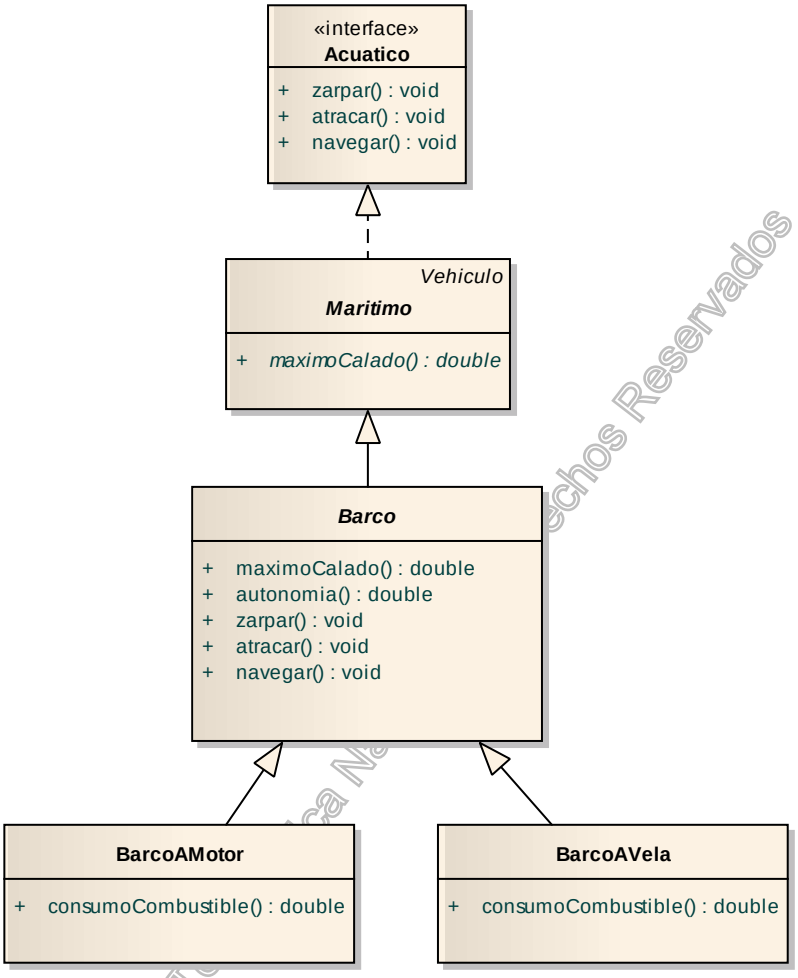
Al igual como se hizo en la sub cadena de herencia a partir de la clase Aereo, se puede tomar otra y realizar el mismo proceso. Por ejemplo, tomando a la clase Maritimo se puede realizar el mismo tipo de refinamiento

Ejemplo



Luego de extraer en una interfaz las acciones en común a otros objetos, el resultado es

Ejemplo



Partiendo de este último ejemplo, se puede ver como llevar al código el refinamiento del diseño. Por ejemplo, la declaración de la interfaz sería de la siguiente manera.

Ejemplo

```
package transporte.refinamiento;

public interface Acuatico {
	void zarpar();
	void atracar();
	void navegar();
}
```

Notar que la implementación de la interfaz omite la declaración del modificador de visibilidad, lo cual no es un requerimiento en ella.

Y la implementación de la misma en la clase abstracta.

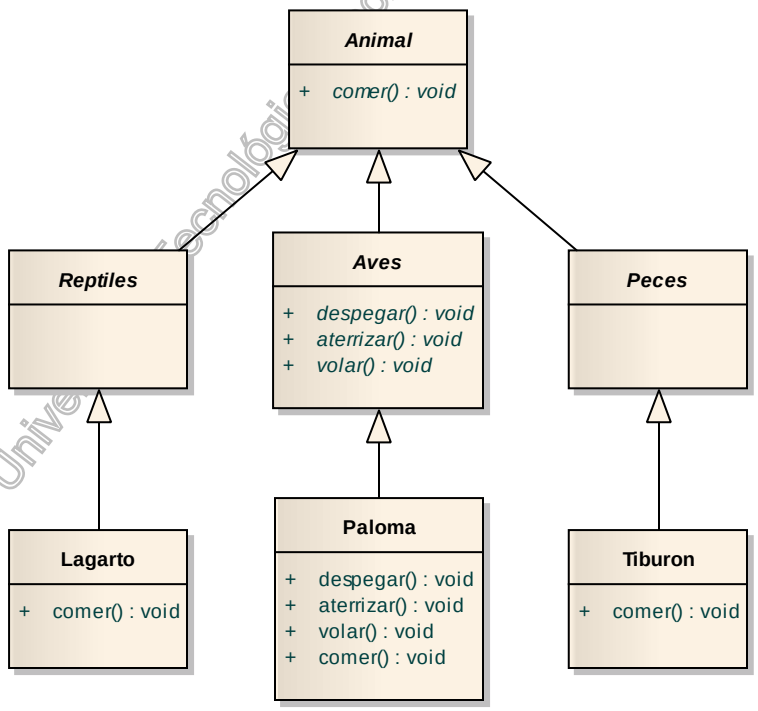
Ejemplo

```
package transporte.refinamiento;
import transporte.Vehiculo;

public abstract class Maritimo extends Vehiculo implements Acuatico{

	public abstract double maximoCalado();

}
```

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.

El conjunto de clases expuesto pertenece a una representación de herencias de clases del tipo Animal.

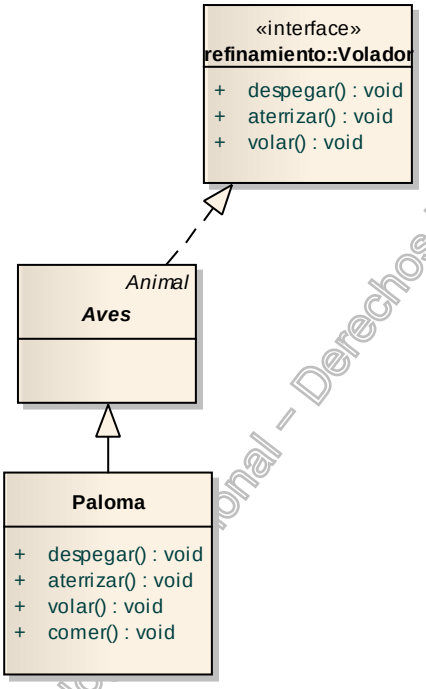
Ejemplo



Refinamiento

Si se observa con detenimiento la clase Aves, se puede realizar un refinamiento no sólo similar al del ejemplo anterior, sino que también se descubre que se puede reutilizar una interfaz ya definida, Volador. Repitiendo el mismo razonamiento que en el ejemplo de Aéreo, las clases quedarían con el siguiente formato

Ejemplo



Revisando el código, la interfaz volador tenía la siguiente forma.

Ejemplo

```
package transporte.refinamiento;

public interface Volador {
	public void despegar();
	public void aterrizar();
	public void volar();
}
```

Su posterior implementación en la clase obliga a importar la interfaz para su utilización en la clase.

Ejemplo

```
package animales.refinamiento;

import transporte.refinamiento.Volador;
```

```
import animales.Animal;

public abstract class Aves extends Animal implements Volador {
}
```

Nota: la clase Aves no define métodos propios a fines de simplificar el ejemplo

En la sub cadena de herencia a partir de la clase Aves, existen otras clases que tiene la obligación de sobrescribir todos los métodos abstractos para poder crear objetos de su tipo. Como la clase Aves implementa la interfaz, los métodos de esta se suman a los definidos como abstractos en la clase como aquellos que se deben rescribir. La siguiente clase "concreta" demuestra el hecho con el método comer, el cual es abstracto desde la clase Animal, y los métodos de la interfaz.

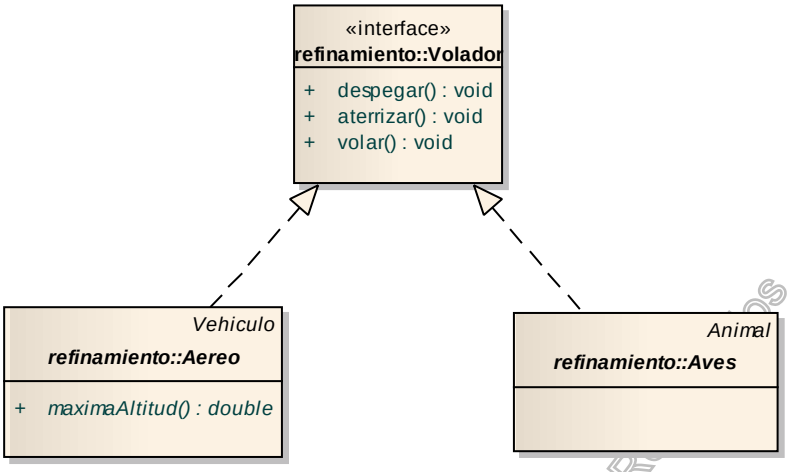
Ejemplo

```
package animales.refinamiento;

public class Paloma extends Aves {

	public void despegar() {
		System.out.println("Paloma despegando");

	}

	public void aterrizar() {
		System.out.println("Paloma aterrizando");
	}

	public void volar() {
		System.out.println("Paloma volando");

	}

	public void comer() {
		System.out.println("Paloma comiendo");
	}

}
```


Relacionando Refinamientos

Todo el proceso anterior permite la unión en un solo gráfico de las clases que implementan la interfaz para ver como se relacionan. Esto puede luego ser aprovechado por programas que quieran implantar el uso de dicha "relación".

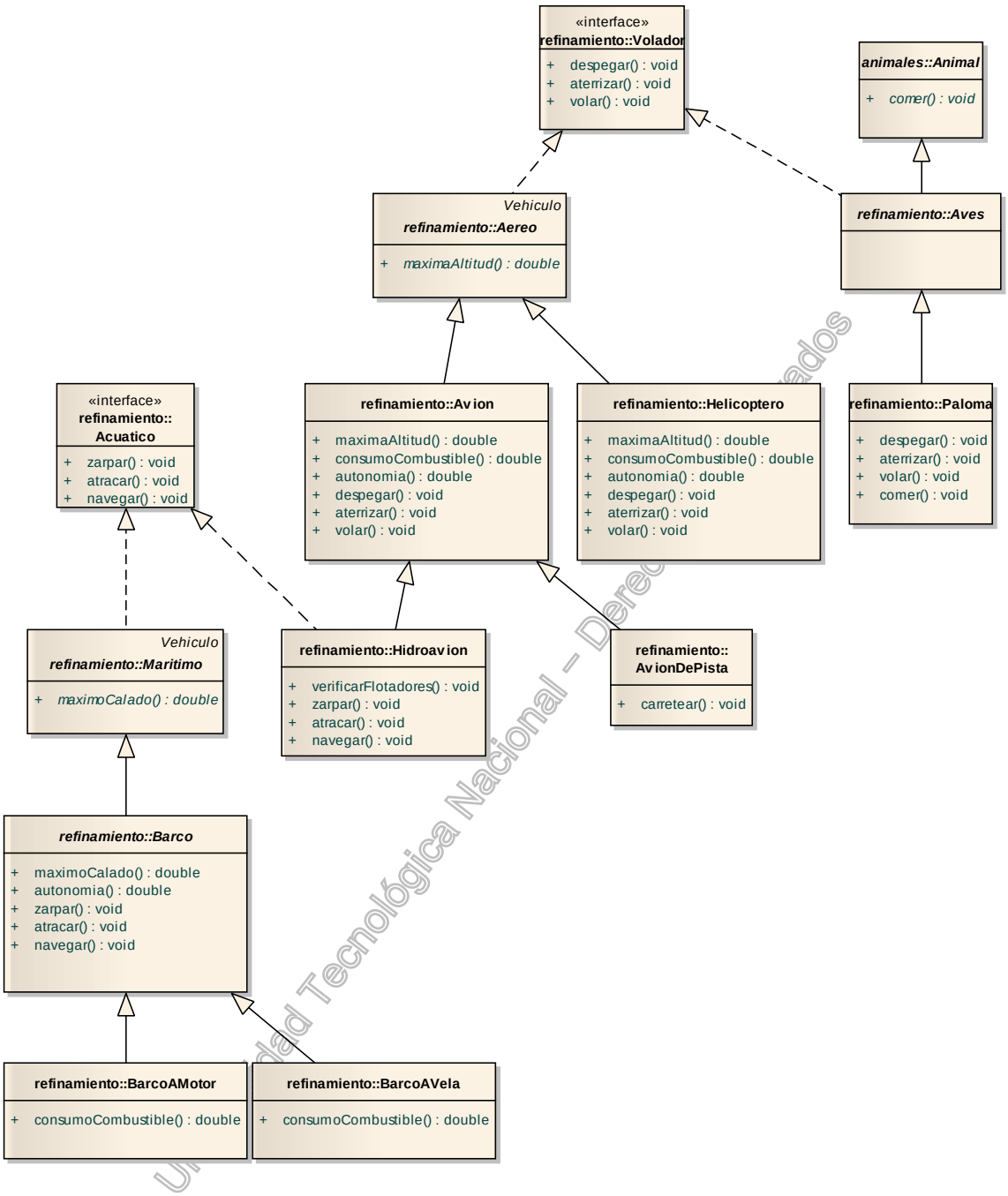
Ejemplo



Integración de Diagramas

Para comprender el formato final de los refinamientos y las cadenas de herencia que se verán afectadas por ellos, se realiza un diagrama que lo refleje. Este tipo de diagramas por lo general asiste en la definición de nuevas clases que utilicen los servicios que las interfaces definen.

Ejemplo



Uso de Interfaces

El diseño de interfaces permite la invocación de todos los métodos que las mismas definen en las clases concretas que los implementan.

Se debe tener cuidado al momento de diseñar la clase porque se pueden usar conceptualmente mal este tipo de relaciones.

El siguiente ejemplo muestra ese caso, en el cual una paloma **supuestamente** pide permiso para aterrizar y despegar. Si bien esto no tiene sentido, demuestra claramente el cuidado que se debe tener al aplicar las relaciones conseguidas por medio de las interfaces, porque el programa en si funciona perfectamente a nivel de código.

Ejemplo

```
package interfaces;

import transporte.refinamiento.Helicoptero;
import transporte.refinamiento.Hidroavion;
import transporte.refinamiento.Volador;
import animales.refinamiento.Paloma;

public class CampoDeAterrizaje {
	public static void main(String[] args) {
		CampoDeAterrizaje cda = new CampoDeAterrizaje();
		Helicoptero h = new Helicoptero();
		Hidroavion ha = new Hidroavion ();
		Paloma p = new Paloma();
		p.comer();
		System.out.println("Autonomía del helicóptero");
		h.autonomia();
		System.out.println("Autonomía del hidroavión");
		ha.autonomia();

		cda.permisoParaAterrizar(h);
		cda.permisoParaAterrizar(ha);
		cda.permisoParaAterrizar(p);

		cda.permisoParaDespegar(h);
		cda.permisoParaDespegar(ha);
		cda.permisoParaDespegar(p);
	}

	public void permisoParaAterrizar(Volador v){
		v.aterrizar();
		//v.autonomia(); Ilegal. No es accesible autonomia()
	}
	public void permisoParaDespegar(Volador v){
		v.despegar();
	}
}
```


Por otro lado, se puede apreciar en el siguiente ejemplo las ventajas brindadas por las interfaces al ser utilizadas correctamente.

Ejemplo

```
package interfaces;

import transporte.refinamiento.Acuatico;
import transporte.refinamiento.BarcoAVela;
```

```
import transporte.refinamiento.Hidroavion;

public class Muelle {
	public static void main(String[] args) {
		Muelle m = new Muelle();
		Hidroavion ha = new Hidroavion();
		BarcoAVela b = new BarcoAVela();

		m.permisoParaAtracar(ha);
		m.permisoParaAtracar(b);

		m.permisoParaZarpar(ha);
		m.permisoParaZarpar(b);
	}

	public void permisoParaZarpar(Acuatico a){
		a.zarpar();
	}
	public void permisoParaAtracar(Acuatico a){
		a.atracar();
	}
}
```

Nota: las interfaces definen la capacidad tecnológica de relacionar más de una clase de distintas cadenas de herencia, ***pero la responsabilidad de la implementación de los métodos y su posterior uso es del programador***

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.

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 esta 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

Ejemplos de Clases Anidadas

En el siguiente ejemplo, se expone el uso de la primera propiedad mostrando el acceso a un atributo de la clase externa (la que anida a otra)

Sin embargo, cuando se quiere utilizar algo de la clase que se encuentra anidada, se tiene que declarar un objeto de la misma y accederlo con notación de punto. La razón es que como se definió previamente, sólo se pueden acceder a los miembros de una clase cuando se define un objeto de la misma.

Ejemplo

```
package anidadas;
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();
          }
}
```

El siguiente ejemplo muestra como resolver ambigüedades entre los nombres de los miembros de la clase. En este caso se tiene tres niveles de visibilidad diferente:

- A nivel de la clase externa
- A nivel de la clase anidada
- A nivel de variable local de un método de la clase anidada

Para resolver los nombres dentro de una clase se utiliza **this**, y esto se mantiene igual en el caso de las clases anidadas, ya que **this** siempre tiene la referencia "del objeto actualmente en ejecución".

Con lo único que se debe tener cuidado es que ahora **this** estará limitado en visibilidad si el objeto esta anidado, por lo tanto se debe resolver la misma con notación de punto para que el código "sepa" si se refiere al **this** de objeto del tipo exterior o al **this** del objeto del tipo de la clase anidada. Esto se resuelve fácilmente con nombres totalmente calificados

Ejemplo

```
package anidadas;

public class Exterior2 {
	private int var;

	public class Anidada {
		private int var;

		public void haceAlgo(int var) {
			var++; // El parámetro local
			this.var++; // atributo del objeto anidado
			Exterior2.this.var++; // atributo del objeto exterior
		}
	}
}
```

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

Ejemplo

```
package anidadas;

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 +
                // " localVar=" + localVar + // 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 localVar (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 una clase como un hecho puntual. No se desea que esas clases estén registradas para otras invocaciones sino que se quiere crearlas sólo cuando existen objetos de ese tipo. 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.

Como ejemplo de una clase anónima, se puede evaluar el caso en que una de estas clases tenga que implementar una interfaz. Como se mencionó anteriormente no se puede crear un objeto del tipo de una interfaz porque sus métodos son abstractos.

En el siguiente ejemplo se sobrescribe el método de la interfaz y se crea el objeto en la misma sentencia

Ejemplo

```
public interface Saludo {
	public static final String DEFAULT = "Adiós";

	public void imprimirSaludo();
}



package anonimas;

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();
	}
```

```
}
```

La variable sAnonimo almacena a un objeto sin un nombre de tipo que lo identifique, pero que sobrescribe al método de la interfaz y utiliza la constante declarada en ella.

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.

Ejemplo

```
package anonimas;

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 igual, ya que se pueden definir como sus elementos a 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 plantear 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.

Ejemplo

```
package enumeraciones.constantes;

public class NivelDeSeguridad {

	public static final int ROL_INICIAL = 0;
	public static final int ROL_USUARIO = 1;
	public static final int ROL_MANEJADOR_DE_RECURSOS = 2;
	public static final int ROL_ADMINISTRADOR = 3;

	private int nivel;
	private int grado;

	public NivelDeSeguridad(int nivel, int grado) {
		super();
		this.nivel = nivel;
		this.grado = grado;
	}

	public String getNombreNivel() {
		String nombre = "";
		switch (nivel) {
		case ROL_INICIAL:
			nombre = "Inicial";
			break;
		case ROL_USUARIO:
			nombre = "Usuario";
			break;
		case ROL_MANEJADOR_DE_RECURSOS:
			nombre = "Manejador de Recursos";
			break;
		case ROL_ADMINISTRADOR:
			nombre = "Administrador";
			break;
		default:
			System.err.println("Rol inválido.");
		}
		return nombre;
	}

	public int getNivel() {
		return nivel;
	}
```

```
	public int getGrado() {
		return grado;
	}
}
```

Sin embargo, en tiempo de ejecución, se pueden generar diferentes errores porque el tipo que define la clase no verifica el buen comportamiento al utilizar las constantes, lo cual determina que no sea un tipo asegurado según los valores que maneja.

Ejemplo

```
package enumeraciones.constantes;

public class VerificaNivelDeSeguridad {
	public static void main(String[] args) {
		NivelDeSeguridad nivel1 = new NivelDeSeguridad(
				NivelDeSeguridad.ROL_ADMINISTRADOR, 2);
		System.out.println("el grado es: " + nivel1.getGrado() +
			" para el rol "+ nivel1.getNombreNivel());

		// Se crea un nivel de seguridad que no existe.
		NivelDeSeguridad nivel2 = new NivelDeSeguridad(47, 2);
		System.out.println("el grado es: " + nivel2.getGrado() +
			" para el rol " + nivel2.getNombreNivel());
	}
}
```

Cuando se utiliza la clase se registra el siguiente comportamiento indeseado en la salida

```
el grado es: 2 para el rol Administrador
Rol inválido.
el grado es: 2 para el rol
```

Este manejo de constantes presenta los siguientes inconvenientes:

- **No mantiene la seguridad de los tipos**: puesto que el atributo nivel es simplemente un tipo `int`, es posible pasarle cualquier valor `int` cuando se necesite especificarlo. Además, se podrían aplicar incluso operaciones aritméticas a dos niveles, lo que no tiene sentido a nivel de diseño.
- **No hay espacio de nombres**: es preciso agregar un prefijo a las constantes de los tipos `int` con una cadena (en este caso, ROL_) para evitar conflictos de nombres con las otras constantes `int`. Esto se determina cuando se diseña la clase pensando que el concepto definido por la constante debe ser único.
- **Inestabilidad**: dado que los tipos enumerados `int` son constantes durante el tiempo de compilación, se compilan en las clases clientes que los utilizan. Si se agrega una nueva constante entre dos constantes existentes o se cambia su orden, es preciso recompilar los clientes. Si no se hace así, seguirán ejecutándose, pero su comportamiento será indeterminado porque varían los valores asignados a las constantes.

- **Salida en pantalla poco informativa**: dado que se trata de valores enteros, todo lo que se ve cuando aparecen en pantalla es un número, lo cual dice poco del valor o el tipo al que representan. Ésta es la razón por la que la clase necesita el método getNombreNivel.

El nuevo tipo enumerado

La versión 5.0 de Java SE incluye una modalidad de tipos enumerados que mantiene la seguridad de los tipos (type safe enum). En el siguiente 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 del tipo.

Ejemplo

```
package enumeraciones.corregido;

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.

Ejemplo

```
package enumeraciones.corregido;

public class NivelDeSeguridad {
	private Nivel nivel;
	private int grado;

	public NivelDeSeguridad(Nivel nivel, int grado) {
		super();
		this.nivel = nivel;
		this.grado = grado;
	}

	public String getNombreNivel() {
		String nombre = "";
		switch (nivel) {
		case INICIAL:
			nombre = "Inicial";
			break;
		case USUARIO:
			nombre = "Usuario";
			break;
```

```
                case MANEJADOR_DE_RECURSOS:
                        nombre = "Manejador de Recursos";
                        break;
                case ADMINISTRADOR:
                        nombre = "Administrador";
                        break;
                default:
                        // No tiene más sentido. Nunca entra acá
                        System.err.println("Rol inválido.");
                }
                return nombre;
        }

        public Nivel getNivel() {
                return nivel;
        }

        public int getGrado() {
                return grado;
        }
}
```

Esto resuelve el problema de seguridad de los tipos que presentaba la antigua versión usando constantes del tipo **final**. El siguiente código muestra el programa que utiliza la clase actualizado. Si se quitan el signo de comentario en las últimas líneas, daría como resultado un error de compilación porque el valor entero 47 no es del tipo Nivel.

Ejemplo

```
package enumeraciones.basicas;

public class VerificaNivelDeSeguridad {
        public static void main(String[] args) {
                NivelDeSeguridad nivel1 = new NivelDeSeguridad(
                                Nivel.ADMINISTRADOR, 2);
                System.out.println(
                        "el grado es: " + nivel1.getGrado() + " para el rol "
                        + nivel1.getNombreNivel());

        // Se crea un nivel de seguridad que no existe.
//      NivelDeSeguridad nivel2 = new NivelDeSeguridad(47, 2);
//      System.out.println("el grado es: " + nivel2.getGrado() + " para el rol "
//                              + nivel2.getNombreNivel());
        }
}
```


Tipos enumerados avanzados

Lamentablemente, la clase NivelDeSeguridad sigue necesitando un método getNombreNivel para retornar a una determinada clase cliente que solicite el nombre del nivel para que resulte fácil de manejar. Si el programa mostrarse el verdadero valor de Nivel, mostraría el nombre simbólico del valor del tipo. Por ejemplo, `Nivel.INICIAL` aparecería como INICIAL. Esto resulta más fácil de manejar para la clase cliente que "0" si necesita el nombre.

Por otro lado, el nombre del nivel no debería formar parte de la clase NivelDeSeguridad sino, más bien, del tipo Nivel. La nueva modalidad de tipos enumerados permite usar atributos y métodos, igual que en las clases normales. El siguiente código muestra una versión mejorada del primer tipo Nivel declarado con un atributo nombre y un método getNombre. Obsérvese cómo se oculta la información de la forma adecuada con modificador de visibilidad **private** y el método de acceso **public**.

Ejemplo

```
package enumeraciones.avanzadas;

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 enum para el valor INICIAL. Los tipos enumerados pueden tener cualquier cantidad de atributos y métodos.

Estas declaraciones de cadenas permiten que no se necesite más un método que deba retornar el nombre del nivel en la clase cliente, cuyo código se simplifica notablemente.

Ejemplo

```
package enumeraciones.avanzadas;

public class NivelDeSeguridad {

	private Nivel nivel;
	private int grado;

	public NivelDeSeguridad(Nivel nivel, int grado) {
		super();
		this.nivel = nivel;
		this.grado = grado;
	}

	public Nivel getNivel() {
		return nivel;
	}
```

```
	public int getGrado() {
		return grado;
	}
}
```

Por último, el siguiente código muestra el programa de prueba modificado, que utiliza el nuevo método getNombre del tipo Nivel. El método getNivel devuelve un valor de nivel y el método getNombre devuelve el atributo nombre del valor actual de Nivel.

Ejemplo

```
package enumeraciones.avanzadas;
public class VerificaNivelDeSeguridad {
	public static void main(String[] args) {
		NivelDeSeguridad nivel1 = new NivelDeSeguridad(
				Nivel.ADMINISTRADOR, 2);
		System.out.println(
			"el grado es: " + nivel1.getGrado() + " para el rol "
			+ nivel1.getNivel().getNombre());
		NivelDeSeguridad nivel2 = new NivelDeSeguridad(Nivel.USUARIO, 2);
		System.out.println(
			"el grado es: " + nivel2.getGrado() + " para el rol "
			+ nivel2.getNivel().getNombre());
	}
}
```

Importaciones estáticas

Cuando una enumeración no se encuentra en el mismo paquete que la clase que la utiliza, es necesario importarla.

Ejemplo

```
package enumeraciones.importaciones;

import enumeraciones.importaciones.estaticas.Nivel;

public class VerificaNivelDeSeguridad {
	public static void main(String[] args) {
		NivelDeSeguridad nivel1 = new NivelDeSeguridad(
				Nivel.ADMINISTRADOR, 2);
		System.out.println(
			"el grado es: " + nivel1.getGrado() + " para el rol "
			+ nivel1.getNivel().getNombre());

		NivelDeSeguridad nivel2 = new NivelDeSeguridad(Nivel.USUARIO, 2);
		System.out.println(
			"el grado es: " + nivel2.getGrado() + " para el rol "
			+ nivel2.getNivel().getNombre());
	}

}
```

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 una función de 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. En el siguiente código se muestra el uso de las importaciones estáticas.

Ejemplo

```
package enumeraciones.importaciones;

import static enumeraciones.importaciones.estaticas.Nivel.*;

public class VerificaNivelDeSeguridad2 {
	public static void main(String[] args) {
		NivelDeSeguridad nivel1 = new NivelDeSeguridad(
				ADMINISTRADOR, 2);
		System.out.println(
			"el grado es: " + nivel1.getGrado() + " para el rol "
			+ nivel1.getNivel().getNombre());

		NivelDeSeguridad nivel2 = new NivelDeSeguridad(USUARIO, 2);
		System.out.println(
			"el grado es: " + nivel2.getGrado() + " para el rol "
			+ nivel2.getNivel().getNombre());
	}
}
```

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.