Tema 1

Programación Orientada a Objetos

Objetos

En la programación orientada a objetos, POO, el problema a resolver se modela mediante componentes de código llamados **objetos** que son abstracciones de los objetos, animados e inanimados, del mundo real. Una **abstracción** es una representación parcial de los atributos y comportamiento de un objeto real. Los **atributos** son las características que definen al objeto y el comportamiento representa lo que el objeto sabe hacer, su funcionalidad. El comportamiento de un objeto es modelado por piezas de código llamados **métodos**. Los atributos y comportamiento representados por el objeto son los que son relevantes al problema que se está modelando. Por ejemplo, supongamos que estamos simulando el comportamiento del consumo de combustible de un automóvil: el objeto automóvil tendrá los atributos: coeficiente de arrastre, cilindrada del motor, aceleración, presión de las llantas, peso, etc. Los métodos que describen su comportamiento son: acelerar, frenar, desplegar velocidad, desplegar nivel de gasolina, etc. El color de la carrocería o el precio no serán atributos ya que no son relevantes en el problema bajo estudio, como tampoco lo serán obtener precio, establecer precio, obtener color, establecer color, etc.

Es posible que un objeto real tenga dos o más abstracciones, es decir, que sea modelado por dos o más objetos dependiendo del problema que se este resolviendo. Por ejemplo, un automóvil en un sistema de inventario tendrá otros atributos como marca, modelo, color, precio, etc., pero no el coeficiente de arrastre ni la capacidad del tanque de combustible. Los métodos son obtener precio, establecer precio, obtener color, establecer color, etc., pero no el acelerar, frenar, etc.

Identificación de los Objetos que Modelan un Problema

El primer paso para modelar un problema usando la POO es la identificación de los objetos que son relevantes para representar el problema. No todos los posibles objetos que podamos hallar son necesarios en nuestro modelo. Los primeros candidatos a ser objetos son los sustantivos (nombres comunes) en el enunciado de un problema. Por ejemplo considere el siguiente problema:

> Un amante de la música y del cine desea catalogar su colección de música y películas. Desea realizar consultas de música por título, autor, género, álbum, intérprete y período; mientras que para sus películas desea hacer consultas por título, género, actor, directo y período.

En este problema, los objetos más obvios son cada una de las canciones y películas que el amante de la música y el cine posee. Los atributos de cada canción son: título, género, intérprete, autor de letra, autor de música, álbum, disquera, duración y fecha; los de cada película son: el título, género, actores principales, director, compañía productora, duración y fecha. Los métodos de cada una de esas canciones y películas nos deben permitir obtener y establecer los valores de cada atributo, por ejemplo, establece nombre, obtén nombre, establece género, obtén género, etc.

Sin embargo en este problema podría, de ser necesario, otros objetos como los intérpretes, autores, actores, directores, géneros, fechas, e inclusive objetos compuestos de objetos como lista de canciones, o listas de actores.

Clases

El segundo paso para modelar un problema usando la POO, es la clasificación de los objetos que son relevantes para representar el problema en grupos de objetos que compartan el mismo tipo de atributos y métodos. La lista de atributos y métodos de cada grupo de objetos (un grupo podría estar formada por un sólo objeto) se conoce como una clase, esto es, una **clase** son los atributos y métodos comunes a un grupo de objetos. Una clase constituye una plantilla con la que se construyen objetos, se crean **instancias** de esa clase.

Para representar una clase, sus atributos y sus métodos podemos utilizar la notación gráfica empleada por los diagramas de clase del lenguaje UML (Lenguaje Unificado de Modelación), que representa a una clase mediante un rectángulo subdividido a su vez en tres rectángulos. En el superior aparece el nombre de la clase, en el central los atributos y en el inferior sus métodos. Por ejemplo, las clases **Cancion** y **Pelicula** del problema anterior estarían representados por los diagramas de la figura 1.1. También se agrega la clase **Genero** utilizada para representar los géneros de las canciones y películas:

Encapsulado

Uno de los principios de la POO es la encapsulación (los otros dos son la herencia y el polimorfismo). La encapsulación es un mecanismo que permite agrupar los datos (atributos) y el código (métodos) que los manipulan y los mantiene alejados de posibles interferencias y usos indebidos. El acceso a los datos y métodos es en forma controlada a través de una **interfaz** bien definida. La base del encapsulado es la clase. Cada atributo y método tiene asociado un modificador de acceso que establece quien tiene acceso a los atributos y métodos de la clase. Los atributos y métodos privados sólo pueden ser

accedidos por el código dentro de la clase. Cualquier otro código fuera de la clase no puede accederlos. Los atributos y métodos públicos son los que pueden ser accedidos desde fuera de la clase y constituyen la interfaz de la clase. En los diagramas de clase, los atributos y métodos privados van precedidos de un signo menos (-) y los públicos por un signo más (+).

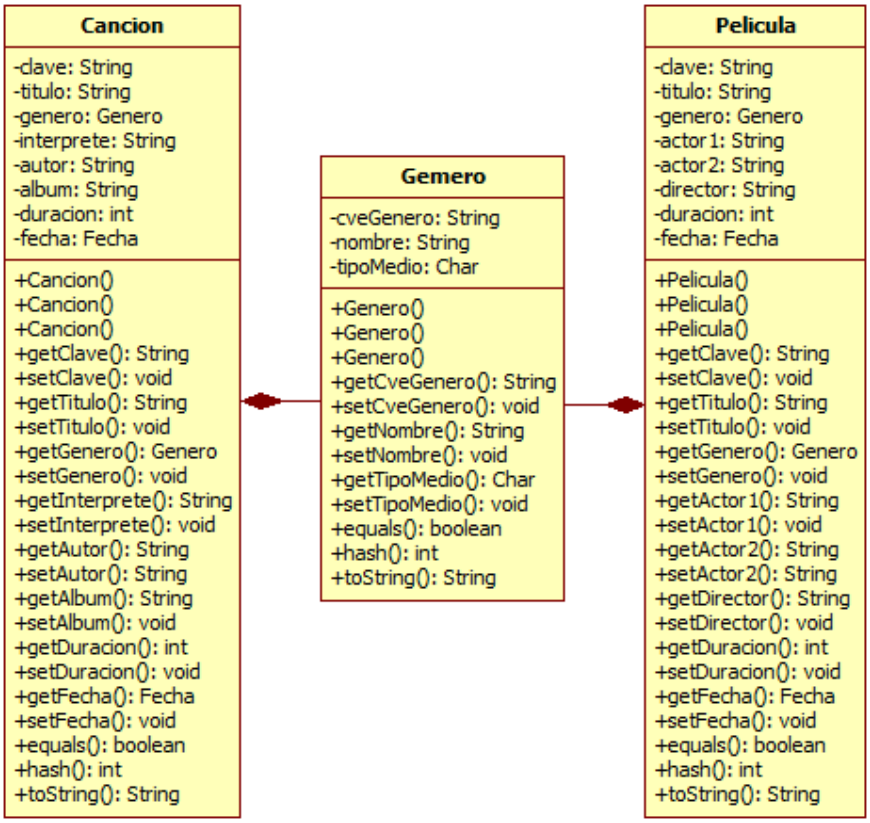

Figura 1.1

Herencia

El segundo principio de la POO es la herencia. El mecanismo de **herencia** permite:

- Que una clase herede las propiedades (atributos) y el comportamiento (métodos) de otra clase, evitando tener que repetir el código que se desea heredar.
- Agrupar en una clase las propiedades (atributos) y comportamiento (métodos) comunes de dos o más clases escribiendo en una sola clase el código común a esas clases. Esas clases heredan de la clase que tienen los atributos y métodos comunes.

La clase de las que heredan otras clases se conoce como superclase y las clases que heredan se llaman subclases. Una subclase es una versión

especializada de una superclase, que hereda todos los atributos y métodos de la superclase y le añade otros.

Por ejemplo, si agrupamos los atributos y métodos de las clases canción y película, del problema sobre el amante de la música y el cine, en la clase **Medio** y de ésta última hacemos que hereden las clases **Cancion** y **Pelicula**, tendríamos el diagrama de clases mostrado en la figura 1.2:

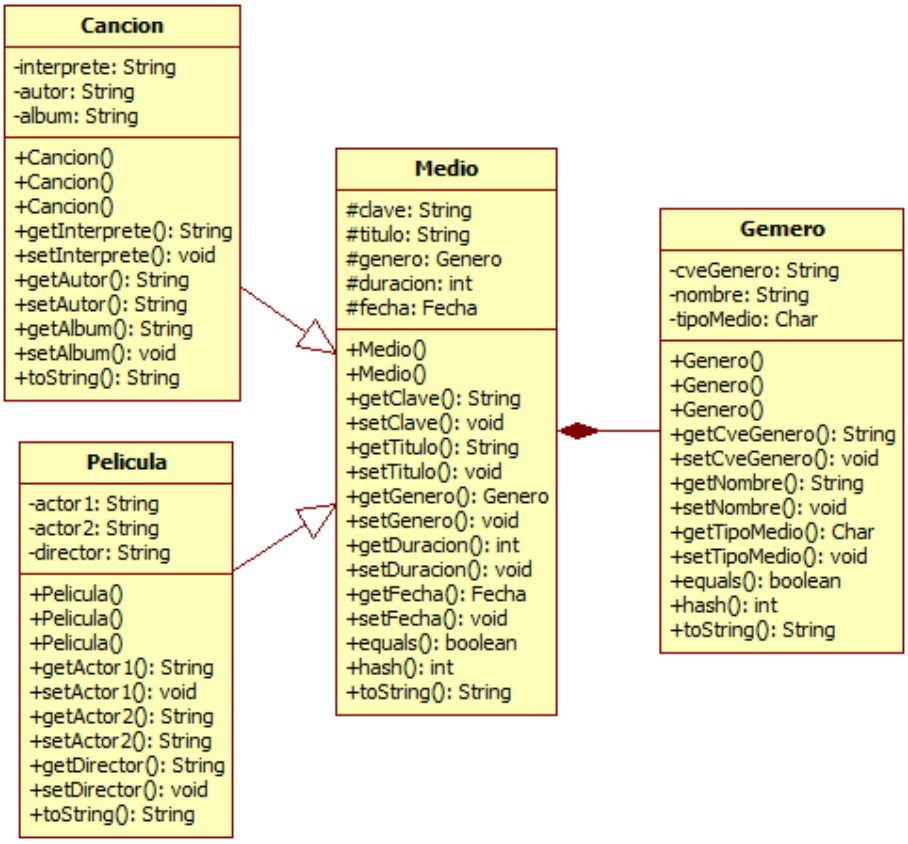

Figura 1.2

Como deseamos que los atributos de la superclase sean visibles dentro de sus subclases le asignamos el modificador de acceso protegido que en los diagramas de clase se denota con el símbolo de número (#). Note, que en los diagramas de clase la herencia se representa mediante una flecha con punta en forma de triángulo apuntando a la superclase.

Codificación de Clases en Java

El código de una clase de Java se almacena en un archivo con el mismo nombre de la clase y con la extensión .java.

Java hace distinción entre mayúsculas y minúsculas.

La sintaxis de una clase en Java es:

```
[package nomPaquete;]

[directivas import]

public class NomClase {
   [declaraciones de atributos]

   [constructor]...
   [método]...
}
```

Donde *nomPaquete* es el nombre del paquete en el que se almacenará el código bytecode de la clase. Un paquete es una carpeta o subdirectorio que contiene una o más clases que están relacionadas.

Los paquetes tienen una estructura jerárquica. Esto es, un paquete puede contener paquetes que a su vez pueden contener paquetes, etc., por lo que *nomPaquete* es la trayectoria al paquete deseado.

Las *directivas import* permiten importar la definición de una clase. La sintaxis de una directiva `import` es:

```
import ruta.NomClase
```

Donde *ruta* es la secuencia de paquetes, separados por puntos, que hay que seguir para llegar a la clase deseada *NomClase.*

Nomclase es el nombre de la clase.

Cada una de las declaraciones de atributos tiene la siguiente sintaxis:

```
[modificadorAcceso] tipo nomAtributo[,nomAtributo]...
```

Donde *modificadorAcceso* es el modificador de acceso del atributo. Los modificadores de acceso se usan para establecer el ámbito de un identificador, es decir en que parte de un programa puede usarse el identificador. Los tipos de modificadores de acceso que se pueden emplear en las declaraciones de atributos y métodos se muestran en la tabla 1.1

tipo es el tipo del atributo y puede ser uno de los tipos predefinidos (**char**, **byte**, **short**, **int**, **long**, **float** o **double**) o una clase.

Tabla 1.1 Modificadores de Acceso para los Atributos y Métodos de una Clase

Modificador de Accesibilidad	Accesible para			
	Clase	**Subclase**	**Paquete**	**Universal**
private	Sí	No	No	No
Por omisión	Sí	No	Sí	No
protected	Sí	Sí	Sí	No
public	Sí	Sí	Sí	Sí

nomAtributo es el nombre del atributo

Constructores

Un constructor de una clase es un método al que se invoca cuando se crea una instancia de esa clase. Realiza toda la inicialización necesaria del nuevo objeto.

- Se puede definir cero, uno o más constructores para una clase.
- Si no se define un constructor, el compilador crea uno por ausencia, el cual no tiene parámetros.
- Todos los constructores se crean por el mecanismo de sobrecarga. Esto es, tienen el mismo nombre que es el mismo de la clase y difieren en el número y tipo de parámetros.
- No se especifica el tipo que regresa el constructor.
- El nombre del constructor es el nombre de la clase.

La sintaxis de un constructor es:

```
public nomClase([lista de parámetros]) {
   [declaraciones de variables locales]

   sentencias
}
```

lista de parámetros es una lista de declaraciones de parámetros separadas por comas (,). Cada una de las declaraciones de parámetros tiene la siguiente sintaxis:

```
tipo nomParametro
```

Donde *tipo* es el tipo del parámetro y puede ser uno de los tipos predefinidos (**char**, **byte**, **short**, **int**, **long**, **float** o **double**) o una clase.

nomParametro es el nombre del parámetro.

Cada una de las declaraciones de variables locales tiene la siguiente sintaxis:

```
tipo nomVariable[,nomVariable]...;
```

Donde *tipo* es el tipo de la variable local y puede ser uno de los tipos predefinidos (**char**, **byte**, **short**, **int**, **long**, **float** o **double**) o una clase. *nomVariable* es el nombre de la variable local.

Las sentencias pueden ser sentencias de expresión o simple, compuestas o bloques y de control.

Métodos

La sintaxis de un método es:

```
[modificadorAcceso] tipo nomMétodo(lista de parámetros) {
   declaraciones de variables locales

   sentencias
}
```

Donde *modificadorAcceso* es el modificador de acceso del método, los modificadores de acceso que pueden tener los métodos se muestran en la tabla 1.

tipo es el tipo del valor regresado por el método y puede ser uno de los tipos predefinidos (**char**, **byte**, **short**, **int**, **long**, **float** o **double**) o una clase.

nomMetodo es el nombre del método.

Los parámetros, las variables locales y las sentencias de los métodos tienen la misma sintaxis que las de los constructores.

Parámetros

Un parámetro es un mecanismo mediante el cual un constructor o un método reciben un valor que requiere para realizar su tarea. Ese valor, llamado **argumento**, le es enviado al constructor o al método al invocarlos.

Por cada parámetro que tenga la definición de un constructor o de un método deberá haber un argumento en su invocación y ese argumento deberá ser del mismo tipo que el parámetro.

Sobrecarga

En una clase podemos tener dos o más métodos o constructores que tengan el mismo nombre y que se distingan por el número y/o tipo de sus parámetros. En este caso se dice que los métodos o constructores están **sobrecargados**. El código de los métodos o constructores son diferentes.

Métodos de Acceso

Si se desea tener acceso a los atributos privados o protegidos de una clase podemos emplear métodos llamados de acceso. Para el atributo

```
tipo nomAtributo;
```

los métodos:

```
public tipo getNomAtributo() {
  return nomAtributo;
}
```

nos regresa el valor del atributo y

```
public void setNomAtributo(tipo nomParametro) {
   nomAtributo = nomparametro;
}
```

permite modificar el valor del atributo.

Ejemplos Sobre Codificación de Clases

Los siguientes listados muestran las codificaciones de las clases **Medio** y **Genero** del ejemplo sobre el amante de la música y el cine.

Medio.java

```
/*
 * Medio.java
 *
 * Creada el 8 de septiembre de 2006, 01:14 PM
 */

package objetosNegocio;

import objetosServicio.Fecha;

/**
 * Esta clase contiene los atributos y métodos comunes a las clases
 * Cancion y Pelicula del programa AmanteMusica
 *
 * @author mdomitsu
 */
public class Medio {
  protected String clave;
  protected String titulo;
  protected Genero genero;
  protected int duracion;
  protected Fecha fecha;

  /**
   * Constructor predeterminado
   */
  public Medio() {
  }

  /**
   * Constructor que inicializa los atributos de la clase
   * @param clave Clave de la canción o película
   * @param titulo Título de la canción o película
   * @param genero Género de la canción o película
   * @param duracion Duración de la canción o película
   * @param fecha Fecha de liberación de la canción o película
   */
  public Medio(String clave, String titulo, Genero genero,
               int duracion, Fecha fecha) {
    this.clave = clave;
    this.titulo = titulo;
    this.genero = genero;
    this.duracion = duracion;
    this.fecha = fecha;
  }

  /**
```

```
   * Regresa la clave de la canción o película
   * @return Clave de la canción o película
   */
  public String getClave() {
    return clave;
  }

  /**
   * Establece la clave de la canción o película
   * @param clave Clave de la canción o película
   */
  public void setClave(String clave) {
    this.clave = clave;
  }

  /**
   * Regresa el título de la canción o película
   * @return Titulo de la canción o película
   */
  public String getTitulo() {
    return titulo;
  }

  /**
   * Establece el título de la canción o película
   * @param titulo Titulo de la canción o película
   */
  public void setTitulo(String titulo) {
    this.titulo = titulo;
  }

  /**
   * Regresa el género de la canción o película
   * @return Género de la canción o película
   */
  public Genero getGenero() {
    return genero;
  }

  /**
   * Establece el género de la canción o película
   * @param genero Género de la canción o película
   */
  public void setGenero(Genero genero) {
    this.genero = genero;
  }

  /**
   * Regresa la duración de la canción o película
   * @return Duración de la canción o película
   */
  public int getDuracion() {
    return duracion;
  }

  /**
   * Establece la duración de la canción o película
   * @param duracion Duración de la canción o película
   */
  public void setDuracion(int duracion) {
    this.duracion = duracion;
```

```
  }

  /**
   * Regresa la fecha de liberación de la canción o película
   * @return Fecha de liberación de la canción o película
   */
  public Fecha getFecha() {
    return fecha;
  }

  /**
   * Establece la fecha de liberación de la canción o película
   * @param fecha Fecha de liberación de la canción o película
   */
  public void setFecha(Fecha fecha) {
    this.fecha = fecha;
  }

  /**
   * Este método compara este medio con el objeto del parámetro
   * @param obj Objeto contra el que se compara este medio
   * @return Veradero si el objeto del parámetro es de la clase Medio
   * y ambos tienen la misma clave, falso en caso contrario.
   */
  @Override
  public boolean equals(Object obj) {
    // Si el parametro es nulo regresa falso
    if (obj == null) {
      return false;
    }

    // Si el parametro no es de la clase Medio regresa falso
    if (getClass() != obj.getClass()) {
      return false;
    }

    final Medio other = (Medio) obj;

    // Regresa verdadero si las dos claves son iguales, falso en caso
    // contrario
    if ((this.clave == null) ? (other.clave != null) :
      !this.clave.equals(other.clave)) {
      return false;
    }

    return true;
  }

  /**
   * Regresa el codigo hash asociado a una instancia de esta clase. El
   * codigo hash es el mismo entero para dos medios que son iguales
   * bajo el metodo equals().
   * @return El codigo hash asociado a una instancia de esta clase
   */
  @Override
  public int hashCode() {
    int hash = 5;

    // Calcula el código hash para este medio en función del
    // código hash de la clave
    hash = 47*hash + (this.clave != null? this.clave.hashCode(): 0);
```

```
    return hash;
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  @Override
  public String toString() {
    return clave + ", " + titulo + ", " + genero .getNombre() + ", " +
           duracion + ", " + fecha;
  }
}
```

Métodos `equals()`, `getClass()`, `hashCode()` y `toString()`

En el tope de la jerarquía de clases de la API de Java se encuentra la clase `Object`, esto es, todas las clases de Java heredan directamente o indirectamente de esta clase. Esta clase define una serie de métodos que todas las clases heredan y que pueden redefinir o sobrescribir. Cuatro de los métodos que tiene la clase `Object` son:

- `boolean equals(Object obj)`: Regresa verdadero si este objeto es "igual" al objeto del parámetro.
- `Class getClass()`: Regresa la clase a la que pertenece este objeto.
- `int hashCode()`: Regresa el código Hash para este objeto. El codigo hash es el mismo entero para dos medios que son iguales bajo el metodo `equals()`.
- `String toString()`: Regresa una cadena con una representación de este objeto.

Las clases `Medio` y `Genero` sobrescriben el método `equals()` para que dos medios, dos canciones, dos películas o dos géneros son iguales si sus claves son iguales. De igual forma, estas clases sobrescriben el método `hashCode()` para que dos medios, dos canciones, dos películas o dos géneros generen el mismo código hash si son iguales. Por último las clases `Medio`, `Cancion`, `Pelicula` y `Genero` sobrescriben el método `toString()` para que sus correspondientes objetos regresen una cadena con la concatenación de los valores de atributos.

Genero.java

```
/*
 * Genero.java
 *
 * Creada el 28 de julio de 2008, 07:18 PM
 */

package objetosNegocio;

/**
 * Esta clase representa el género de una canción o película
```

```
 *
 * @author mdomitsu
 */
public class Genero {
  private String cveGenero;
  private String nombre;
  private char tipoMedio;

  /**
   * Constructor por omisión
   */
  public Genero() {
  }

  /**
   * Constructor que inicializa los atributos de la clase
   * @param cveGenero Clave del género de la canción y película
   * @param nombre Género de la canción o película
   * @param tipoMedio Tipo del medio (canción o película) del género.
   *              'C' = canción, 'P' = película
   */
  public Genero(String cveGenero, String nombre, char tipoMedio) {
    this.cveGenero = cveGenero;
    this.nombre = nombre;
    this.tipoMedio = tipoMedio;
  }

  /**
   * Constructor que inicializa el atributo cveGenero
   *
   * @param cveGenero Clave del género de la canción y película
   */
  public Genero(String cveGenero) {
    this(cveGenero, null, ' ');
  }

  /**
   * Regresa la clave del género de la canción o película
   *
   * @return La clave del género de la canción o película
   */
  public String getCveGenero() {
    return cveGenero;
  }

  /**
   * Establece la clave del género de la canción o película
   *
   * @param cveGenero Clave del género de la canción o película
   */
  public void setCveGenero(String cveGenero) {
    this.cveGenero = cveGenero;
  }

  /**
   * Regresa el género de la canción o película
   * @return El género de la canción o película
   */
  public String getNombre() {
    return nombre;
  }
```

```
/**
 * Establece el género de la canción o película
 *
 * @param tipoMedio Género de la canción o película
 */
public void setNombre(String nombre) {
  this.nombre = nombre;
}

/**
 * Regresa el tipo del medio (canción o película)
 * @return El tipo del medio (canción o película)
 */
public char getTipoMedio() {
  return tipoMedio;
}

/**
 * Establece el tipo del medio (canción o película)
 *
 * @param tipoMedio Tipo del Medio (canción o película)
 */
public void setTipo(char tipoMedio) {
  this.tipoMedio = tipoMedio;
}

/**
 * Este método compara este genero con el objeto del parámetro
 * @param obj Objeto contra el que se compara este genero
 * @return Veradero si el objeto del parámetro es de la clase Genero
 * y ambos tienen la misma clave, falso en caso contrario.
 */
@Override
public boolean equals(Object obj) {
  // Si el parametro es nulo regresa falso
  if (obj == null) {
    return false;
  }

  // Si el parametro no es de la clase Genero regresa falso
  if (getClass() != obj.getClass()) {
    return false;
  }

  final Genero other = (Genero) obj;

  // Regresa verdadero si las dos claves son iguales, falso en caso
  // contrario
  if ((this.cveGenero == null)? (other.cveGenero != null):
    !this.cveGenero.equals(other.cveGenero)) {
    return false;
  }

  return true;
}

/**
 * Regresa el codigo hash asociado a una instancia de esta clase. El
 * codigo hash es el mismo entero para dos generos que son iguales
 * bajo el metodo equals().
```

```
   * @return El codigo hash asociado a una instancia de esta clase
   */
  @Override
  public int hashCode() {
    int hash = 7;

    // Calcula el código hash para este genero en función del
    // código hash de la clave
    hash = 71 * hash +
      (this.cveGenero != null? this.cveGenero.hashCode(): 0);

    return hash;
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  @Override
  public String toString() {
    return cveGenero + ", " + nombre + ", " + tipoMedio;
  }
}
```

Convenciones de Codificación de Clases

Note que por convención, en la codificación de las clases:

- Los nombres de los paquetes empiezan con una letra minúscula.
- Los nombres de las clases empiezan con una letra mayúscula.
- Los nombres de los atributos empiezan con una letra minúscula.
- Los nombres de los métodos empiezan con una letra minúscula.
- En los identificadotes compuestos por dos o más palabras, las palabras siguientes a la primera empiezan con una letra mayúscula. Una alternativa es separar las palabras con el carácter (_).
- Los identificadores de los métodos de acceso empiezan con las palabras **`get`** y **`set`** seguidas del nombre del atributo con la primera letra de éste en mayúsculas.

Codificación de SubClases en Java

La sintaxis para definir una subclase es la siguiente:

```
[package nomPaquete;]

[directivas import]

public class NomClase extends NomSuperClase{
   [declaraciones de atributos]

   [constructor]...
   [método]...
}
```

La palabra reservada **`extends`** establece que la clase dada por *`NomClase`* hereda de la clase *`NomSuperClase`*. Los atributos, constructores y métodos declarados y definidos aquí son propios de la subclase. Los atributos y métodos de la superclase también forman parte de la subclase y los puede acceder directamente a menos que sus modificadores de acceso sean privados.

Constructores en las Subclases

Si la superclase tiene uno o más constructores y ninguno de ellos es el constructor por ausencia, entonces todos los constructores de una subclase de esa superclase deben de invocar a uno de los constructores de la superclase y esa invocación debe ser la primera instrucción del constructor. Lo anterior se debe a que cuando se crea un objeto de una subclase primero debe llamarse al constructor de la superclase.

La invocación de uno de los constructores de la superclase tiene la siguiente sintaxis:

```
super(lista de los argumentos);
```

La palabra reservada **`super`** es una referencia a la clase padre de la clase en que se usa dicha palabra.

Ejemplos Sobre Codificación de Subclases

Los listados siguientes muestran el código de las clases **Cancion** y **Pelicula**, las cuales heredan de la clase Medio.

Cancion.java

```
/*
 * Cancion.java
 *
 * Creada el 9 de septiembre de 2006, 12:43 AM
 */

package objetosNegocio;

import objetosServicio.Fecha;

/**
 * Esta clase contiene los atributos y métodos de una canción del
 * programa AmanteMusica
 *
 * @author mdomitsu
 */
public class Cancion extends Medio {
  private String interprete;
  private String autor;
  private String album;

  /**
   * Constructor predeterminado
```

```
   */
  public Cancion() {
    super();
  }

  /**
   * Constructor que inicializa los atributos de la clase
   * @param clave Clave de la canción
   * @param titulo Título de la canción
   * @param genero Género de la canción
   * @param interprete Intérprete de la canción
   * @param autor Autor de la canción
   * @param album Álbum de la canción
   * @param duracion Duración de la canción
   * @param fecha Fecha de liberación de la canción
   */
  public Cancion(String clave, String titulo, Genero genero,
                 String interprete, String autor, String album,
                 int duracion, Fecha fecha) {
    super(clave, titulo, genero, duracion, fecha);
    this.interprete = interprete;
    this.autor = autor;
    this.album = album;
  }

  /**
   * Constructor que inicializa el atributo clave
   * @param clave Clave de la canción
   */
  public Cancion(String clave) {
    this(clave, null, null, null, null, null, 0, null);
  }

  /**
   * Regresa el intérprete de la canción
   * @return Intérprete de la canción
   */
  public String getInterprete() {
    return interprete;
  }

  /**
   * Establece el intérprete de la canción
   * @param interprete Intérprete de la canción
   */
  public void setInterprete(String interprete) {
    this.interprete = interprete;
  }

  /**
   * Regresa el autor de la canción
   * @return Autor de la canción
   */
  public String getAutor () {
    return autor;
  }

  /**
   * Establece el autor de la canción
   * @param autorLetra Autor de la canción
   */
```

```
  public void setAutor(String autor) {
    this.autor = autor;
  }

  /**
   * Regresa el álbum de la canción
   * @return Álbum de la canción
   */
  public String getAlbum() {
    return album;
  }

  /**
   * Establece el álbum de la canción
   * @param album Álbum de la canción
   */
  public void setAlbum(String album) {
    this.album = album;
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  @Override
  public String toString() {
    return super.toString() + ", " + interprete + ", " + autorLetra +
           ", " + autorMusica + ", " + album + ", " + disquera;
  }
}
```

Pelicula

```
/*
 * Pelicula.java
 *
 * Creada el 9 de septiembre de 2006, 12:50 AM
 */

package objetosNegocio;

import objetosServicio.Fecha;

/**
 * Esta clase contiene los atributos y métodos de una película
 * del programa AmanteMusica
 *
 * @author mdomitsu
 */
public class Pelicula extends Medio {
  private String actor1;
  private String actor2;
  private String director;

  /**
   * Constructor predeterminado
   */
  public Pelicula() {
  }

  /**
   * Constructor que inicializa los atributos de la clase
```

```
   * @param clave Clave de la película
   * @param titulo Título de la película
   * @param genero Género de la película
   * @param actor1 Primer actor de la película
   * @param actor2 Segundo actor de la película
   * @param director Director de la película
   * @param duracion Duración de la película
   * @param fecha Fecha de liberación de la película
   */
  public Pelicula(String clave, String titulo , Genero genero,
                  String actor1, String actor2, String director,
                  int duracion, Fecha fecha) {
    super(clave, titulo, genero, duracion, fecha);
    this.actor1 = actor1;
    this.actor2 = actor2;
    this.director = director;
  }

  /**
   * Constructor que inicializa el atributo clave
   * @param clave Clave de la canción
   */
  public Pelicula(String clave) {
    this(clave, null, null, null, null, null, 0, null);
  }

  /**
   * Regresa el primer actor de la película
   * @return Primer actor de la película
   */
  public String getActor1() {
    return actor1;
  }

  /**
   * Establece el primer actor de la película
   * @param actor1 Primer actor de la película
   */
  public void setActor1(String actor1) {
    this.actor1 = actor1;
  }

  /**
   * Regresa el segundo actor de la película
   * @return Segundo actor de la película
   */
  public String getActor2() {
    return actor2;
  }

  /**
   * Establece el segundo actor de la película
   * @param actor2 Segundo actor de la película
   */
  public void setActor2(String actor2) {
    this.actor2 = actor2;
  }

  /**
   * Regresa el director actor de la película
   * @return Director actor de la película
```

```
   */
  public String getDirector() {
    return director;
  }

  /**
   * Establece el director actor de la película
   * @param director Director actor de la película
   */
  public void setDirector(String director) {
    this.director = director;
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  @Override
  public String toString() {
    return super.toString() + ", " + actor1 + ", " + actor2 + ", " +
           actor3 + ", " + actor4 + ", " + director + ", " +
           productora;
  }
}
```

Polimorfismo

El tercer principio de la POO es el polimorfismo. El **polimorfismo** permite que una subclase redefina el comportamiento de un método de su superclase. Esa redefinición o **sobreescritura** de métodos se da cuando la superclase y la subclase tienen un método que tiene la misma **firma** (el mismo nombre y la misma lista de parámetros) y el mismo tipo (Tipo del valor que regresa), pero difieren en el cuerpo del método.
Si en la subclase invocamos a un método sobrescrito, el método invocado será el de la subclase ya que el método de la superclase queda oculto por su homónimo de la subclase ya que tienen el mismo nombre. Si queremos invocar al método sobrescrito de la superclase podemos hacerlo usando la siguiente sintaxis:

super.*nomMetodoSuperClase***(***lista de argumentos***)**

Como un ejemplo de polimorfismo tenemos al método `equals()`,`hashCode()` y `toString()` de las clases `Medio`, `Cancion`, `Pelicula` y `Genero`.

Clases y Métodos Finales

Si no deseamos que de una clase se hereden clases, debemos declarar esa clase como final, utilizando el **modificador final**. La sintaxis de una **clase final** es la siguiente:

```
[package nomPaquete;]

[directivas import]

public final class NomClase {
   [declaraciones de atributos]

   [constructor]...
   [método]...
}
```

Por ejemplo, la clase java.util.Math de la API de Java, está declarada como:

```
public final class Math {
   public static final double PI = 3.141592653589793d;
   ...
   public static double sqrt(double a) {...}
   ...
}
```

Si deseamos que un método de una superclase no sea sobrescrito en una de sus subclases, debemos declarar ese método como final, usando el **modificador final**. La sintaxis de un **método final** es la siguiente:

```
[modificadorAcceso] final tipo nomMétodo(lista de parámetros) {
   declaraciones de variables locales

   sentencias
}
```

Creación de Objetos

Ya se mencionó que una clase constituye una plantilla con la que se construyen objetos, se crean **instancias** de esa clase. La forma más común de crear un objeto es mediante el operador **new**. El operador **new** crea una instancia de una clase y devuelve una referencia a esa instancia. La sintaxis de la creación de un objeto usando el operador **new** es:

```
new constructor;
```

Donde *constructor* es uno de los constructores de la clase. Si la definición de la clase no provee un constructor, se utiliza el constructor por ausencia. Por ejemplo las siguientes líneas de código crean dos objetos de tipo Canción:

```
new Cancion();

new Cancion("CBB0001", "The long and winding way",
      new Genero("GCB0001", "Balada" , 'C'), "The Beatles",
      "John Lennon", "Let it be", 194, new Fecha(24, 3, 1970));
```

Referencias a Objetos

Ya se mencionó que el tipo de un atributo, de un parámetro o de una variable local puede ser una clase. En estos casos se dice que el atributo, parámetro o variable es una referencia. Una **referencia** es un identificador asociado a una instancia de clase (objeto) y nos sirve para referirnos a ese objeto.

Para que un atributo o una variable local hagan referencia a un objeto, debemos asignárselo. La sintaxis de la asignación es la siguiente:

```
{nomAtributo | nomVariable} = objeto;
```

Podemos crear un objeto y al mismo tiempo asignárselo a un atributo o variable. La sintaxis es la siguiente:

```
{nomAtributo | nomVariable} = new constructor;
```

Por último, podemos combinar la declaración de un atributo o variable local, la creación de un objeto y sus asociación con la siguiente sintaxis:

```
tipo {nomAtributo | nomVariable} = new constructor;
```

Por ejemplo:

```
Cancion cancion1 = new Cancion("CBB0001",
   "The long and winding way",
   new Genero("GCB0001", "Balada", 'C'), "The Beatles",
   "John Lennon", "Let it be", 194, new Fecha(24, 3, 1970));

Fecha fecha1 = new Fecha(1, 8, 1965);

Genero genero1 = new Genero("GCB0001", "Balada" , 'C');

Cancion cancion2, cancion3;

cancion2 = cancion1;

cancion3 = new Cancion("CBB0002", "Yesterday", genero1,
   "The Beatles", "Paul McCartney", "Help!", 123, fecha1);
```

En el ejemplo anterior se crean dos objetos de tipo **Cancion** y se les asigna a las referencias `cancion1` y `cancion3`. Por otro lado, en la sentencia:

```
cancion2 = cancion1;
```

le estamos asignado a la referencia `cancion2` el mismo objeto asignado a la referencia `cancion1`. Esto es, tanto `cancion1` como `cancion2` hacen referencia al mismo objeto. No a objetos diferentes.
La asociación entre un parámetro y el objeto al que hace referencia ocurre en el momento de la invocación del método. Por ejemplo, en el código anterior tenemos dos ejemplos de cómo el parámetro fecha del constructor de la clase **Cancion** se asocia a un objeto. En el primer caso, en la creación del objeto

`cancion1`, el parámetro fecha hace referencia al objeto `new Fecha(24, 3, 1970)` creado en el momento de la invocación. En el segundo caso, en la creación del objeto `cancion3`, el parámetro `fecha` hace referencia al objeto fecha1, creado previamente.

Referencias y Herencia

A una referencia a una superclase podemos asignarle una instancia de una se sus subclases. Por ejemplo:

```
Medio medio;
Cancion cancion1 = new Cancion(“CBB0001”,
   “The long and winding way”,
   new Genero(“GCB0001”, “Balada” ,‘C’), “The Beatles”,
   “John Lennon”, “Let it be”, 194, new Fecha(24, 3, 1970));

medio = cancion1;
```

En el ejemplo anterior, la clase **Cancion** hereda de la clase **Medio** y podemos ver que a la referencia `medio` que es del tipo **Medio** le estamos asignando un objeto de tipo **Cancion** que hereda del tipo **Medio**.

Por otro lado a una referencia de una subclase no podemos asignarle una instancia de su clase padre.

```
Cancion cancion2 = medio;
```

Generaría un error de ejecución. Una excepción.

Acceso a los Atributos y Métodos de una Instancia de una Clase

Para acceder a los atributos y métodos públicos de una clase (o con modificador de acceso predeterminado desde otra clase en el mismo paquete), se utiliza la siguiente sintaxis:

nomObjeto.*nomAtributo*

o

nomObjeto.**nomMetodo(***lista de argumentos***)**

Por ejemplo:

```
canción1.setTitulo(“Norwegian Wood”);

String artista = cancion1.getInterprete();
```

Acceso a los Atributos y Métodos y Herencia

Si a una referencia de una superclase le asignamos una instancia de una subclase, sólo podremos acceder a los atributos y métodos de la superclase. Por ejemplo, dadas las declaraciones y asignaciones siguientes:

```
Medio medio;
Cancion cancion1 = new Cancion("CBB0001",
   "The long and winding way",
   new Genero("GCB0001", "Balada" ,'C'), "The Beatles",
   "John Lennon", "Let it be", 194, new Fecha(24, 3, 1970));

medio = cancion1;
```

Las siguiente sentencias sería válida:

```
medio.setTitulo("Norwegian Wood");
```

pero la siguiente no:

```
String artista = medio.getInterprete();
```

ya que `getInterprete()` no es un método de **Medio** sino de **Cancion**. En lugar de ello tendríamos que escribir:

```
String artista = ((Cancion)medio).getInterprete();
```

para convertir la referencia `medio` de ser una referencia a **Medio** a una referencia a **Cancion**.

Como ejemplo de creación de objetos, referencia a objetos, acceso a atributos y métodos de un objeto se muestra la clase `Prueba1` que se utiliza para probar las clases del paquete `objetosnegocio` del programa sobre el amante de la música y el cine. En el método main() de esta clase se crean instancias de las clases `Cancion`, `Pelicula` y `Genero` y se acceden a sus métodos de acceso para obtener y modificar sus atributos.

Prueba1.java

```
/*
 * Prueba1.java
 *
 * Creada el 8 de septiembre de 2006, 12:21 PM
 */

package pruebas;

import objetosServicio.Fecha;
import objetosNegocio.*;

/**
 * Esta clase se utiliza para probar las clases del proyecto
 * amanteMusicaObjNeg
 *
 * @author mdomitsu
 */
public class Prueba1 {
```

```
  /**
   * Creates a new instance of Prueba1
   */
  public Prueba() {
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // Se crean tres géneros de canciones
    Genero genero1 = new Genero("GCB0001", "Balada", 'C');
    Genero genero2 = new Genero("GCB0002", "Bossanova", 'C');
    Genero genero3 = new Genero("GCR0003", "Rock", 'C');

    // Se crean tres géneros de películas
    Genero genero4 = new Genero("GPD0001", "Drama", 'P');
    Genero genero5 = new Genero("GPC0002", "Ciencia Ficción", 'P');
    Genero genero6 = new Genero("GPC0003", "Comedia", 'P');

    // Se despliegan los datos del género 1
    System.out.println("Género 1: " + genero1);
    // Se despliegan los datos del género 2
    System.out.println("Género 2: " + genero2);
    // Se despliegan los datos del género 3
    System.out.println("Género 3: " + genero3);
    // Se despliegan los datos del género 4
    System.out.println("Género 4: " + genero4);
    // Se despliegan los datos del género 5
    System.out.println("Género 5: " + genero5);
    // Se despliegan los datos del género 6
    System.out.println("Género 6: " + genero6);

    // Se crean tres canciones
    Cancion cancion1 = new Cancion("CBB0001",
       "The long and winding road", genero1, "The Beatles",
       "John Lennon", "Let it be", 3, new Fecha(24, 3, 1970));
    Cancion cancion2 = new Cancion("CSD0002", "Garota de Ipanema",
       genero2, "Los Indios Tabajaras", "Antonio Carlos Jobim",
       "Bossanova Jazz Vol. 1", 3, new Fecha(1, 12, 1970));
    Cancion cancion3 = new Cancion("CSB0003", "Desafinado", genero2,
       "Joao Gilberto", "Joao Gilberto", "Bossanova Jazz Vol. 1",
       3, new Fecha(3, 12, 1980));

    // Se despliegan los datos de la canción 1
    System.out.println("Cancion 1: " + cancion1);
    // Se despliegan los datos de la canción 2
    System.out.println("Cancion 2: " + cancion2);
    // Se despliegan los datos de la canción 3
    System.out.println("Cancion 3: " + cancion3);

    // Se despliega el titulo de la canción 1
    System.out.println("Titulo de la canción 1: "
       + cancion1.getTitulo());
    // Se despliega la fecha de la canción 2
    System.out.println("Fecha de la canción 2: "
       + cancion2.getFecha());

    // Se cambia el autor de la canción 3
    cancion3.setAutor("Antonio Carlos Jobim");
    // Se despliegan los datos de la canción 3
```

```
    System.out.println("Cancion 3: " + cancion3);

    // Se crean dos películas
    Pelicula pelicula1 = new Pelicula("PED0001", "Casa Blanca",
       genero3, "Humphrey Bogart", "Ingrid Bergman",
      "Michael Curtiz", 102, new Fecha(1, 1, 1942));
    Pelicula pelicula2 = new Pelicula("PCF0002",
       "2001 Space Odyssey", genero4, "Keir Dullea", "Gary Lockwood",
       "Stanley Kubrick", 141, new Fecha(1, 1, 1968));

    // Se despliegan los datos de la película 1
    System.out.println("Pelicula 1: " + pelicula1);
    // Se despliegan los datos de la película 2
    System.out.println("Pelicula 2: " + pelicula2);

    // Se despliega el titulo de la película 1
    System.out.println("Titulo de la película 1: "
       + pelicula1.getTitulo());
    // Se despliega el género de la película 2
    System.out.println("Género de la película 2: "
       + pelicula2.getGenero().getNombre());
  }
}
```

Métodos y Clases Abstractas

Muchas veces, la superclase describe en forma general el comportamiento que tendrán sus subclases. Esto es, debido a su generalidad, no todos los métodos de una superclase pueden definirse (sólo se declaran) y su definición se posterga para sus subclases. Esos métodos de la superclase que sólo se declaran se conocen como abstractos. La sintaxis de una declaración de un método abstracto es:

```
[modificadorAcceso] abstract tipo nomMétodo(lista de
                                            parámetros);
```

Una clase que contiene al menos un método abstracto también es una clase abstracta. Su sintaxis es:

```
[package nomPaquete;]

[directivas import]

public abstract class NomClase {
   [declaraciones de atributos]

   [constructor]...
   [método]...
   [declaración de un método abstracto]...
}
```

No se puede instanciar una clase abstracta, esto es no podemos crear objetos de una clase abstracta.

Si una clase que hereda de una clase abstracta no implementa todos los métodos abstractos de su superclase debe ser declarada abstracta a su vez.

Ejemplos Sobre Métodos y Clases Abstractas

Como ejemplo de métodos y clases abstractas tenemos el siguiente problema:

> Una fábrica de silos para granos produce silos de tres tipos: Cilíndricos, cónicos y esféricos. Al fabricante de silos le interesa saber cuánta lámina debe emplear para cada silo y la capacidad del silo. Para ello debe conocer la superficie que tendrá cada silo y su volumen.

El diagrama de clases para el programa a construirse se muestra en la figura 1.3:

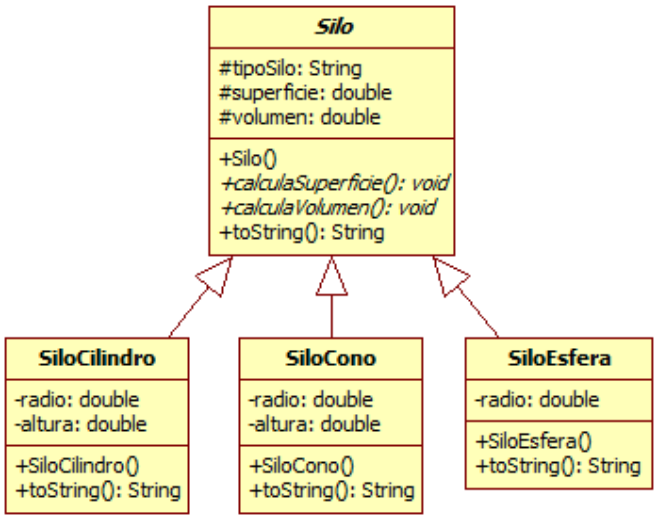

Figura 1.3. Diagrama de clases del problema del fabricante de Silos

- El atributo `tipoSilo` de la clase Silo es una cadena: “Silo Cilíndrico”, “Silo Cónico” o “Silo Esférico”.
- El constructor de la clase Silo inicializa el atributo `tipoSilo` al valor de su parámetro.
- Los métodos `calculaSuperficie()` y `calculaVolumen` de la clase `Silo` son abstractos, sus definiciones están en sus clases hijas.
- El método `toString()` de la clase `Silo` regresa una cadena con el tipo de silo.
- Los constructores de las clases SiloCilindro, SiloCono y SiloEsfera le dan nombre al silo, inicializan los atributos con las dimensiones del silo.
- Los métodos `calculaSuperficie()` y `calculaVolumen()` de las clases SiloCilindro, SiloCono y SiloEsfera calculan la superficie y el volumen de sus respectivos silos empleando las fórmulas de la tabla 1.2:
- Los métodos `toString()`de las clases SiloCilindro, SiloCono y SiloEsfera regresan una cadena con el tipo de silo, sus dimensiones, su superficie y su volumen.

Tabla 1.2

	Superficie	Volumen
Silo Cilíndrico	$2\pi r(r+h)$	$\pi r^2 h$
Silo Cónico	$\pi r(r+\sqrt{r^2+h^2})$	$\frac{1}{3}\pi r^2 h$
Silo Esférico	$4\pi r^2$	$\frac{4}{3}\pi r^3$

La codificación de las clases Silo, SiloCilindro, SiloCono y SiloEsfera se muestra a continuación:

Silo.java

```
/*
 * Silo.java
 *
 * Creada el 5 de octubre de 2005, 12:21 PM
 */

package silos;

/**
 * Esta clase abstracta es la clase padre de las clases SiloCilindro,
 * SiloCono y SiloEsfera
 *
 * @author mdomitsu
 */
public abstract class Silo {
  protected String tipoSilo;
  protected double superficie;
  protected double volumen;

  /**
   * Constructor. Inicializa el atributo tipoSilo.
   * @param tipoSilo Tipo de silo: "Cilindro", "Cono", "Esfera"
   */
  public Silo(String tipoSilo) {
    this.tipoSilo = tipoSilo;
  }

  public abstract void calculaSuperficie();
  public abstract void calculaVolumen();

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
   public String toString() {
     return tipoSilo;
   }
}
```

SiloCilindro.java

```
/*
 * SiloCilindro.java
 *
 * Creada el 5 de octubre de 2005, 12:33 PM
 */
```

```
package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCilindro
 *
 * @author mdomitsu
 */
public class SiloCilindro extends Silo {
  private double radio;
  private double altura;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo cilíndrico
   * @param altura Altura del silo cilíndrico
   */
  public SiloCilindro(double radio, double altura) {
    // Llama al constructor de la clase padre: Silo
    super("Silo Cilíndrico");
    this.radio = radio;
    this.altura = altura;
  }

  /**
   * Calcula la superficie del cilindro
   */
  public void calculaSuperficie() {
    superficie = 2 * Math.PI * radio * (radio + altura);
  }

  /**
   * Calcula el volumen del cilindro
   */
  public void calculaVolumen() {
    volumen = Math.PI * radio * radio * altura;
  }

  /**
   * Genera una cadena con los valores de los atributos de la clase
   * @return Una cadena con los valores de los atributos de la clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = "
        + altura + ", superficie = " + superficie + ", volumen = "
        + volumen;
  }
}
```

SiloCono.java

```
/*
 * SiloCono.java
 *
 * Creada el 5 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCono
 *
 * @author mdomitsu
```

```
 */
public class SiloCono extends Silo {
  private double radio;
  private double altura;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo coníco
   * @param altura Altura del silo coníco
   */
  public SiloCono(double radio, double altura) {
    // Llama al constructor de la clase padre: Silo
    super("Silo Cónico");
    this.radio = radio;
    this.altura = altura;
  }

  /**
   * Calcula la superficie del cono
   */
  public double calculaSuperficie() {
    superficie = Math.PI * radio * (radio
           + Math.sqrt(radio * radio + altura * altura));
  }

  /**
   * Calcula el volumen del cono
   */
  public void calculaVolumen() {
    volumen = Math.PI * radio * radio * altura / 3;
  }

  /**
   * Genera una cadena con los valores de los atributos de la clase
   * @return Una cadena con los valores de los atributos de la clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = "
           + altura + ", superficie = " + superficie + ", volumen = "
           + volumen;
  }
}
```

SiloEsfera.java

```
/*
 * SiloEsfera.java
 *
 * Creada el 5 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloEsfera
 *
 * @author mdomitsu
 */
public class SiloEsfera extends Silo {
  private double radio;

  /**
```

```
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo esférico
   */
  public SiloEsfera(double radio) {
    // Llama al constructor de la clase padre: Silo
    super("Silo Esférico");
    this.radio = radio;
  }

  /**
   * Calcula la superficie de la esfera
   */
  public double calculaSuperficie() {
    superficie = 4 * Math.PI * radio * radio;
  }

  /**
   * Calcula el volumen de la esfera
   */
  public double calculaVolumen() {
    volumen = 4 * Math.PI * radio * radio * radio/3;
  }

  /**
   * Genera una cadena con los valores de los atributos de la clase
   * @return Una cadena con los valores de los atributos de la clase
*/
  public String toString() {
    return super.toString() + ", radio = " + radio + ", superficie = "
           + superficie + ", volumen = " + volumen;
  }
}
```

Para probar las clases anteriores podemos emplear la siguiente clase de prueba:

PruebaSilo.java

```
/*
 * PruebaSilo.java
 *
 * Creada el 5 de octubre de 2005, 12:45 PM
 */

package silos;

/**
 * Esta clase se utiliza para probar las clases SiloCilindro, SiloCono
 * y SiloEsfera
 *
 * @author mdomitsu
 */
public class PruebaSilo {

  /**
   * Método main() en el que se invocan a los métodos de las clases
   * SiloCilindro, SiloCono y SiloEsfera para probarlos
   * @param argumentos Los argumentos en la línea de comando
   */
  public static void main(String[] args) {
    PruebaSilo pruebaSilo1 = new PruebaSilo();
```

```
    Silo silo[]=new Silo[5];

    // Crea 5 silos de diferente tipo
    silo[0]=new SiloCilindro(1.0, 1.0);
    silo[1]=new SiloEsfera(1.0);
    silo[2]=new SiloCono(1.0, 1.0);
    silo[3]=new SiloCilindro(2.0, 1.0);
    silo[4]=new SiloCilindro(1.0, 2.0);

    // Para cada silo
    for (int i = 0; i < 5; i++) {
      // Calcula la superficie del silo
      silo[i].calculaSuperficie();

      // Calcula el volumen del silo
      silo[i].calculaVolumen();
    }

    // Para cada silo
    for (int i = 0; i < 5; i++)
      // Escribe los valores de sus atributos
      System.out.println(silo[i]);
  }
}
```

En el código de la clase de prueba hay varios puntos a notar:

- El arreglo `silo` se declara del tipo de la clase padre: `Silo`.
- A los elementos del arreglo `silo` se le asignan referencias a objetos de las clases hijas de `Silo`.
- Para calcular la superficie y el volumen de cada silo se invoca a los métodos `calculaSuperficie()` y `calculaVolumen()` de la clase `Silo`. Sin embargo gracias al mecanismo de polimorfismo, el compilador convierte esa llamada a una llamada al método de la clase del objeto almacenado en el elemento.

Para desplegar los valores de los atributos de los silos, al método `println()` se le pasan referencias a los diferentes silos. De nuevo, gracias al mecanismo de poliforfismo, la invocación del método `println()` no invoca al método `toString()` de la clase Silo sino a los métodos `toString()` de las clases hijas correspondientes al tipo del objeto creado.

Si ejecutamos el programa anterior obtendremos el siguiente resultado:

```
Silo Cilíndrico, radio = 1.0, altura = 1.0, superficie =
12.566370614359172, volumen = 3.141592653589793
Silo Esférico, radio = 1.0, superficie = 12.566370614359172,
volumen = 4.1887902047863905
Silo Cónico, radio = 1.0, altura = 1.0, superficie =
7.584475591748159, volumen = 1.0471975511965976
Silo Cilíndrico, radio = 2.0, altura = 1.0, superficie =
37.69911184307752, volumen = 12.566370614359172
Silo Cilíndrico, radio = 1.0, altura = 2.0, superficie =
18.84955592153876, volumen = 6.283185307179586
```

Atributos y Métodos Estáticos

Cuando creamos varias instancias de una misma clase, cada instancia tiene su propia copia de cada atributo. Sin embargo hay ocasiones en las que deseamos que de un atributo de una clase sólo haya una copia y que todas las instancias de esa clase compartan ese atributo. En esos casos debemos declarar al atributo como estático usando el **modificador static.** La sintaxis para un **atributo estático** es la siguiente:

```
[modificadorAcceso] static tipo nomAtributo
```

Por ejemplo suponga que en el ejemplo de la fábrica de silos, deseáramos llevar la cuenta de las instancias de silos creados. Podríamos definir un atributo estático llamado contadorSilos y hacer que el constructor de la clase incrementara en uno su valor cada vez que se creara una instancia de la clase, ver el ejemplo sobre Atributos y Métodos Estáticos en la siguiente sección.

A un atributo puede aplicarse los modificadores **static** y **final** al mismo tiempo. Por ejemplo, el atributo `PI` de la clase Math:

```
public final class Math {
   public static final double PI = 3.141592653589793d;
   ...
}
```

No se requiere instanciar una clase para usar sus atributos estáticos. Podemos acceder a un atributo estático mediante la siguiente sintaxis:

```
nomClase.nomAtributoEstatico
```

Por ejemplo, en el siguiente código note que no se creo un objeto del tipo Math para acceder a su atributo `PI`:

```
area = Math.PI*radio*radio;
```

Una clase también puede tener métodos estáticos. Un **método estático** se declara con el **modificador static**, usando la siguiente sintaxis

```
[modificadorAcceso] static tipo nomMétodo(lista de parámetros) {
   declaraciones de variables locales

   sentencias
}
```

Por ejemplo, el método `sqrt()` de la clase Math es un método estático:

```
public final class Math {
     ...
   public static double sqrt(double a) {...}
}
```

Un **método estático** no requiere de la instanciación de la clase para su utilización. Por ejemplo, en el siguiente código note que no se creo un objeto del tipo Math para acceder a su método `sqrt()`:

```
y = Math.sqrt(x);
```

Dado que los atributos y métodos estáticos no requieren de la instanciación de la clase para su uso se les conoce también como **atributos de clase** y **métodos de clase**, respectivamente. Por otro lado, los atributos y métodos declarados sin el modificador static si requieren de una instancia de clase y por lo tanto se les conocen como **atributos de instancia** y **métodos de instancia**, respectivamente.

Los métodos estáticos tienen ciertas restricciones:

- Sólo pueden llamar a otros métodos estáticos.
- Sólo pueden acceder a atributos estáticos
- No pueden llamar a las referencias `this` o `super`.

Ejemplos Sobre Atributos y Métodos Estáticos

Como ejemplo de atributos y métodos estáticos modificaremos el ejemplo sobre la fábrica de silos vista en la sección anterior, para que incluya el cálculo del costo de cada silo, dependiendo del calibre (espesor) de la lámina usada y del tipo de base empleada: sencilla o reforzada. El diagrama de clases para el programa a construirse se muestra en la figura 1.4:

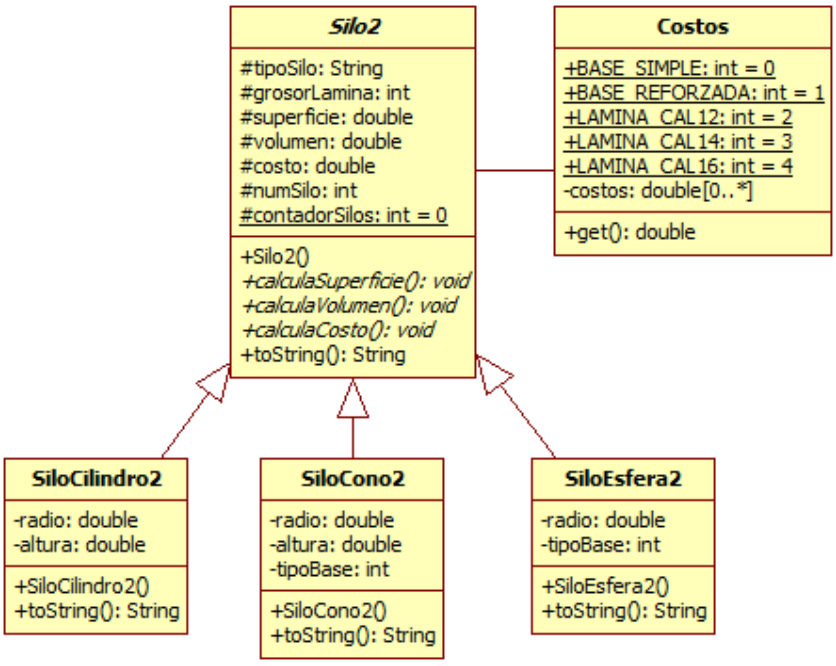

Figura 1.4. Diagrama de clases del problema del fabricante de Silos

- La clase Costos permite obtener el costo de los componentes (costo de las láminas por metro cuadrado y el costo de las bases) empleadas en la construcción de los silos. Los costos están almacenados en el atributo `costos`
- Para acceder a los diferentes costos se emplean las constantes estáticas: `BASE_SIMPLE`, `BASE_REFORZADA`, `LAMINA_CAL12`, `LAMINA_CAL14`, `LAMINA_CAL16`, como parámetros del método `get()`, el cual regresa el costo correspondiente del arreglo costos.
- El constructor de la clase Silo2 inicializa el atributo `tipoSilo` y `grosorLamina` al valor de sus parámetros.
- Los métodos `calculaSuperficie()`, `calculaVolumen` y `calculaCosto()` de la clase Silo2 son abstractos, sus definiciones están en sus clases hijas.
- El método `toString()` de la clase `Silo2` regresa una cadena con el número de silo, el tipo de silo y el grososr de la lámina.
- Los constructores de las clases SiloCilindro2, SiloCono2 y SiloEsfera2 le dan nombre al silo, inicializan los atributos con las dimensiones del silo, el grosor de la lámina y el tipo de base.
- Los métodos `calculaSuperficie()` y `calculaVolumen` de las clases SiloCilindro2, SiloCono2 y SiloEsfera2 calculan la superficie y el volumen de sus respectivos silos empleando las fórmulas de la tabla 1.2:
- El método `calculaCosto()` de las clases SiloCilindro2, SiloCono2 y SiloEsfera2 calculan el costo de sus respectivos silos empleando las fórmulas de la tabla 1.3:

Tabla 1.3

	Costo
Silo Cilíndrico	superficie * costo de la lámina por metro cuadrado
Silos Cónico y Esférico	superficie * costo de la lámina por metro cuadrado + costo de la base

- El método toString() para que regrese una cadena con el tipo de silo, sus dimensiones, grosor de la lámina, tipo de base, su superficie, su volumen y su costo.

La codificación de las clases Costos, Silo2, SiloCilindro2, SiloEsfera2 y SiloCono2 se muestra a continuación:

Costos.java

```
/*
 * Costos.java
 *
 * Creada el 5 de octubre de 2005, 12:36 PM
 */

package silos;

/**
 * Esta clase permite consultar los costos de las componentes de los
 * silos
 *
 * @author mdomitsu
 */
```

```
public class Costos {
  // Indices de los costos en el arreglo costos
  public static final int BASE_SIMPLE = 0;
  public static final int BASE_REFORZADA = 1;
  public static final int LAMINA_CAL12 = 2;
  public static final int LAMINA_CAL14 = 3;
  public static final int LAMINA_CAL16 = 4;

  // Arreglo con los costos
  private static final double costos[] = {600.0, 900.0,
                                          35.0, 45.0, 60.0};

  /**
   * Regresa el costo del componente del silo
   * @param item Indice del componente del que se desea el costo
   * @return El costo del componente del silo
   */
  public static double get(int componente) {
    return costos[componente];
  }
}
```

Silo2.java

```
/*
 * Silo2.java
 *
 * Creada el 6 de octubre de 2005, 12:36 PM
 */

package silos;

/**
 * Esta clase abstracta es la clase padre de las clases SiloCilindro2,
 * SiloCono2 y SiloEsfera2
 *
 * @author mdomitsu
 */
public abstract class Silo2 {
  protected String tipoSilo;
  protected int grosorLamina;
  protected double superficie;
  protected double volumen;
  protected double costo;
  protected int numSilo;
  protected static int contadorSilos = 0;

  /**
   * Constructor. Inicializa el atributo tipoSilo e incrementa el
   * contador de silos en uno cada vez que se crea un silo. También
   * le asigna el valor de ese contador al atributo numSilo para
   * numerar cada silo.
   * @param tipoSilo Tipo de silo: "Cilindro", "Cono", "Esfera"
   * @param grosorLamina Grosor de la lámina del silo cilíndrico
   */
  public Silo2(String tipoSilo, int grosorLamina) {
    // Inicializa los atributos
    this.tipoSilo = tipoSilo;
    this.grosorLamina = grosorLamina;

    // Incrementa el contador de silos creados
    contadorSilos++;
```

```
    // Le asigna un número a este silo
    numSilo = contadorSilos;
  }

  public abstract void calculaSuperficie();
  public abstract void calculaVolumen();
  public abstract void calculaCosto();

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase   */
  public String toString() {
    return numSilo + ": " + tipoSilo + ", Grosor lámina: "
            + grosorLamina;
  }
}
```

SiloCilindro2.java

```
/*
 * SiloCilindro2.java
 *
 * Creada el 6 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCilindro2
 *
 * @author mdomitsu
 */
public class SiloCilindro2 extends Silo2 {
  private double radio;
  private double altura;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo cilíndrico
   * @param altura Altura del silo cilíndrico
   * @param grosorLamina Grosor de la lámina del silo cilíndrico
   */
  public SiloCilindro2(double radio, double altura,
                       int grosorLamina) {
    // Llama al constructor de la clase padre: Silo2
    super("Silo Cilíndrico", grosorLamina);

    // Inicializa los atributos
    this.radio = radio;
    this.altura = altura;
  }

  /**
   * Calcula la superficie del cilindro
   */
  public void calculaSuperficie() {
    superficie = 2*Math.PI*radio*(radio+altura);
  }

  /**
   * Calcula el volumen del cilindro
```

```
   */
  public void calculaVolumen() {
    volumen = Math.PI*radio*radio*altura;
  }

  /**
   * Calcula el precio del silo cilíndrico
   */
  public void calculaCosto() {
    costo = superficie * Costos.get(grosorLamina);
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = "
           + altura + ", superficie = " + superficie + ", volumen = "
           + volumen + ", costo: " + costo;
  }
}
```

SiloCono2.java

```
/*
 * SiloCono2.java
 *
 * Creada el 6 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCono2
 *
 * @author mdomitsu
 */
public class SiloCono2 extends Silo2 {
  private double radio;
  private double altura;
  private int tipoBase;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo cónico
   * @param altura Altura del silo cónico
   * @param tipoBase Tipo de la base del silo cónico
   * @param grosorLamina Grosor de la lámina del silo cónico
   */
  public SiloCono2(double radio, double altura, int tipoBase,
                   int grosorLamina) {
    // Llama al constructor de la clase padre: Silo2
    super("Silo Cónico", grosorLamina);

    // Inicializa los atributos
    this.radio = radio;
    this.altura = altura;
    this.tipoBase = tipoBase;
  }

  /**
   * Calcula la superficie del cono
```

```
   */
  public void calculaSuperficie() {
    superficie = Math.PI*radio*(radio + Math.sqrt(radio*radio +
                 altura*altura));
  }

 /**
   * Calcula el volumen del cono
   */
  public void calculaVolumen() {
    volumen = Math.PI*radio*radio*altura/3;
  }

  /**
   * Calcula el precio del silo cónico
   */
  public void calculaCosto() {
    costo = Costos.get(tipoBase) + superficie *
            Costos.get(grosorLamina);
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase   */
  public String toString() {
    return super.toString() + ", tipo base = + tipoBase + ", radio = "
           + radio + ", altura = " + altura + ", superficie = "
           + superficie + ", volumen = " + volumen + ", costo: "
           + costo;
  }
}
```

SiloEsfera2.java

```
/*
 * SiloEsfera2.java
 *
 * Creada el 6 de octubre de 2005, 12:33 PM
 */
package silos;

/**
 * Esta clase permite crear objetos de tipo SiloEsfera2
 *
 * @author mdomitsu
 */
public class SiloEsfera2 extends Silo2 {
  private double radio;
  private int tipoBase;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo esférico
   */
  public SiloEsfera2(double radio, int tipoBase, int grosorLamina) {
    // Llama al constructor de la clase padre: Silo2
    super("Silo Esférico", grosorLamina);

    // Inicializa los atributos
    this.radio = radio;
    this.tipoBase = tipoBase;
  }
```

```
  /**
   * Calcula la superficie de la esfera
   */
  public void calculaSuperficie() {
    superficie = 4*Math.PI*radio*radio;
  }

  /**
   * Calcula el volumen de la esfera
   */
  public void calculaVolumen() {
    volumen = 4*Math.PI*radio*radio*radio/3;
  }

  /**
   * Calcula el precio del silo esférico
   */
  public void calculaCosto() {
    costo = Costos.get(tipoBase) + superficie *
            Costos.get(grosorLamina);
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase   */
  public String toString() {
    return super.toString()+ ", tipo base = + tipoBase + ", radio = "
           + radio + ", superficie = " + superficie + ", volumen = "
           + volumen + ", costo: " + costo;
  }
}
```

Para probar las clases anteriores se implementa la siguiente clase de prueba:

PruebaSilo2

```
/*
 * PruebaSilo2.java
 *
 * Creada el 6 de octubre de 2005, 12:45 PM
 */

package silos;

/**
 * Esta clase se utiliza para probar las clases SiloCilindro2,
 * SiloCono2 y SiloEsfera2
 *
 * @author mdomitsu
 */
public class PruebaSilo2 {

  /**
   * Método main() en el que se invocan a los métodos de las clases
   * SiloCilindro2, SiloCono2 y SiloEsfera2 para probarlos
   * @param argumentos Los argumentos en la línea de comando
   */
  public static void main(String[] args) {
    PruebaSilo2 pruebaSilo2 = new PruebaSilo2();
```

```
    Silo2 silo[] = new Silo2[5];

    // Crea 5 silos de diferente tipo
    silo[0] = new SiloCilindro2(1.0, 1.0, Costos.LAMINA_CAL12);
    silo[1] = new SiloEsfera2(1.0, Costos.BASE_SIMPLE,
                              Costos.LAMINA_CAL12);
    silo[2] = new SiloCono2(1.0, 1.0, Costos.BASE_SIMPLE,
                            Costos.LAMINA_CAL12);
    silo[3] = new SiloCilindro2(2.0, 1.0, Costos.LAMINA_CAL14);
    silo[4] = new SiloCilindro2(1.0, 2.0, Costos.LAMINA_CAL16);

    // Para cada silo
    for (int i = 0; i < 5; i++) {
        // Calcula la superficie del silo
        silo[i].calculaSuperficie();

        // Calcula el volumen del silo
        silo[i].calculaVolumen();

        // Calcula el costo del silo
        silo[i].calculaCosto();
    }

    // Para cada silo
    for (int i = 0; i < 5; i++)
       // Escribe los valores de sus atributos
      System.out.println(silo[i]);

    // Escribe el número de silos creados
    System.out.println("Silos creados: " + Silo2.contadorSilos);
  }
}
```

Interfaces

Una clase abstracta es aquella que contiene uno o más métodos abstractos y que un método abstracto es aquel que sólo se declara pero no se implementa; la implementación del método debe de hacerse en una clase hija de esa clase abstracta. Una clase abstracta puede tener métodos que tengan tanto su declaración como su implementación. Java expande el concepto de clase abstracta con el concepto de interfaz. Una interfaz contiene la declaración de métodos más no contiene la implementación de ninguno de ellos. Una interfaz declara un conjunto de métodos que una clase debe implementar, esto es establece un determinado comportamiento que la clase que la implementa debe exhibir.

- Una interfaz sólo puede tener atributos estáticos finales y métodos abstractos.
- Aunque todos los métodos de una interfaz son abstractos, no es necesario declarar que lo son usando el modificador `abstract`. Tampoco la interfaz se declara abstracta.
- Los métodos declarados en una interfaz son públicos.
- Una interfaz puede extender (heredar) de una o más interfaces. A esta interfaz se le conoce como subinterfaz y contiene la declaración de los

métodos de las interfaces que extiende más las declaraciones de los métodos que ella misma declara.

- Una clase puede implementar una o más interfaces.
- Una clase puede heredar de una clase e implementar una o más interfaces.
- Una clase que implementa una o más interfaces debe implementar todos los métodos de las interfaces que implementa.
- Una clase que implementa una subinterfaz debe implementar todos los métodos de la subinterfaz y de las interfaces padre.
- Un tipo interfaz es un como un tipo de una clase. Podemos:

    - Declarar variables de un tipo interfaz. A esa variable se le debe asignar un objeto de una clase que implemente dicha interfaz.
    - Declarar un parámetro de un método que sea de un tipo interfaz. Al invocar al método debemos pasarle un objeto de una clase que implemente dicha interfaz.
    - Podemos especificar que el tipo de un método sea de un tipo interfaz. El método deberá regresar un objeto de una clase que implemente dicha interfaz.

Al igual que con las clases, El código de una interfaz de Java se almacena en un archivo con el mismo nombre de la clase y con la extensión .java. Su sintaxis es:

```
[package nomPaquete;]

[directivas import]

public interface NomInterfaz {
   [declaraciones de atributos estáticos finales]

   [declaraciones de métodos]...
}
```

La sintaxis de una subinterfaz es:

```
[package nomPaquete;]

[directivas import]

public interface NomInterfaz extends nomInterfazPadre1[,
                                     nomInterfazPadre2]... {
   [declaraciones de atributos estáticos finales]

   [declaraciones de métodos de la subinterfaz]...
}
```

La sintaxis de una clase que implementa una o más interfaces es:

```
[package nomPaquete;]

[directivas import]

public class NomClase implements NomInterfaz1[,
```

```
                                    NomInterfaz2]... {
   [declaraciones de atributos]

   [constructor]...
   [método de esta clase]...
   [implementación de método de las interfaces]...
}
```

La sintaxis de una clase que hereda de otra clase e implementa una o más interfaces es:

```
[package nomPaquete;]

[directivas import]

public class NomClase extends nomClasePadre
                      implements NomInterfaz1[,
                                 NomInterfaz2]... {
   [declaraciones de atributos]

   [constructor]...
   [método de esta clase]...
   [implementación de método de las interfaces]...
}
```

Ejemplos Sobre Interfaces

Como ejemplo de interfaces modificaremos el ejemplo sobre la fabrica de silos vista en la sección anterior, para que tenga una interfaz llamada `ISilo` con las declaraciones de los métodos `calculaSuperficie()`, `calculaVolumen()` y `calculaCosto()`.

El diagrama de clases para el programa a construirse se muestra en la figura 1.5:

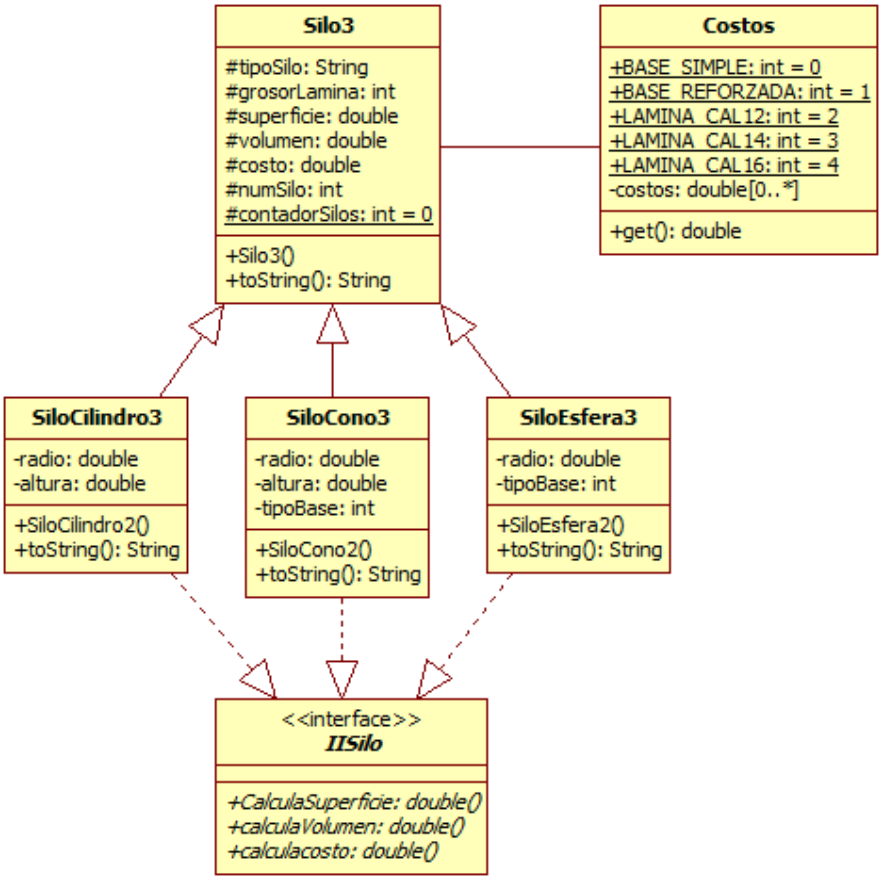

Figura 1.5. Diagrama de clases del problema del fabricante de Silos

La codificación de las clase Costos es la misma del ejemplo anterior. La codificación de la interfaz ISilo y de las clases Silo3, SiloCilindro3, SiloEsfera3 y SiloCono3 se muestra a continuación:

ISilo.java

```
/*
 * ISilo.java
 *
 * Creada el 7 de octubre de 2005, 12:36 PM
 */

package silos;

/**
 * Esta interfaz declara los métodos que implementarán las clases
 * SiloCilindro3, SiloCono3 SiloEsfera3, SiloCilindro4, SiloCono4 y
 * SiloEsfera4
 *
 * @author mdomitsu
 */
public interface ISilo {

  /**
   * Calcula la superficie de un silo
```

```
   */
  public void calculaSuperficie();

  /**
   * Calcula el volumen de un silo
   */
  public void calculaVolumen();

  /**
   * Calcula el costo de un silo
   */
  public void calculaCosto();
}
```

Silo3.java

```
/*
 * Silo3.java
 *
 * Creada el 7 de octubre de 2005, 12:36 PM
 */

package silos;

/**
 * Esta clase es la clase padre de las clases SiloCilindro3,
 * SiloCono3 y SiloEsfera3
 *
 * @author mdomitsu
 */
public class Silo3 {
  protected String tipoSilo;
  protected double superficie;
  protected double volumen;
  protected double costo;
  protected int grosorLamina;
  protected int numSilo;
  protected static int contadorSilos = 0;

  /**
   * Constructor. Inicializa el atributo tipoSilo e incrementa el
   * contador de silos en uno cada vez que se crea un silo.
   * @param tipoSilo Tipo de silo: "Cilindro", "Cono", "Esfera"
   * @param grosorLamina Grosor de la lámina del silo cilíndrico
   */
  public Silo3(String tipoSilo, int grosorLamina) {
    // Inicializa los atributos
    this.tipoSilo = tipoSilo;
    this.grosorLamina = grosorLamina;

    // Incrementa el contador de silos creados
    contadorSilos++;

    // Le asigna un número a este silo
    numSilo = contadorSilos;
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  public String toString() {
```

```
     return numSilo + ": " + tipoSilo + ", Grosor lámina: "
            + grosorLamina;
  }
}
```

SiloCilindro3.java

```
/*
 * SiloCilindro3.java
 *
 * Creada el 7 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCilindro3
 *
 * @author mdomitsu
 */
public class SiloCilindro3 extends Silo3 implements ISilo {
  private double radio;
  private double altura;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo cilíndrico
   * @param altura Altura del silo cilíndrico
   * @param grosorLamina Grosor de la lámina del silo cilíndrico
   */
  public SiloCilindro3(double radio, double altura,
                       int grosorLamina) {
    // Llama al constructor de la clase padre: Silo3
    super("Silo Cilíndrico", grosorLamina);

    // Inicializa los atributos
    this.radio = radio;
    this.altura = altura;
  }

  /**
   * Calcula la superficie del cilindro
   */
  public void calculaSuperficie() {
    superficie = 2*Math.PI*radio*(radio+altura);
  }

  /**
   * Calcula el volumen del cilindro
   */
  public void calculaVolumen() {
    volumen = Math.PI*radio*radio*altura;
  }

  /**
   * Calcula el precio del silo cilíndrico
   */
  public void calculaCosto() {
    costo = superficie * Costos.get(grosorLamina);
  }

  /**
```

```
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = "
           + altura + ", superficie = " + superficie + ", volumen = "
           + volumen + ", costo: " + costo;
  }
}
```

SiloCono3.java

```
/*
 * SiloCono3.java
 *
 * Creada el 7 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloCono3
 *
 * @author mdomitsu
 */
public class SiloCono3 extends Silo3 implements ISilo {
  private double radio;
  private double altura;
  private int tipoBase;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo cónico
   * @param altura Altura del silo cónico
   * @param tipoBase Tipo de la base del silo cónico
   * @param grosorLamina Grosor de la lámina del silo cónico
   */
  public SiloCono3(double radio, double altura, int tipoBase,
                   int grosorLamina) {
    // Llama al constructor de la clase padre: Silo3
    super("Silo Cónico", grosorLamina);

    // Inicializa los atributos
    this.radio = radio;
    this.altura = altura;
    this.tipoBase = tipoBase;
  }

  /**
   * Calcula la superficie del cono
   */
  public void calculaSuperficie() {
    superficie = Math.PI*radio*(radio + Math.sqrt(radio*radio +
                 altura*altura));
  }

 /**
   * Calcula el volumen del cono
   */
  public void calculaVolumen() {
    volumen = Math.PI*radio*radio*altura/3;
  }
```

```
  /**
   * Calcula el precio del silo cónico
   */
  public void calculaCosto() {
    costo = Costos.get(tipoBase) + superficie *
            Costos.get(grosorLamina);
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", altura = "
           + altura + ", superficie = " + superficie + ", v = "
           + volumen + ", Costo: " + costo;
  }
}
```

SiloEsfera3.java

```
*
 * SiloEsfera3.java
 *
 * Creada el 7 de octubre de 2005, 12:33 PM
 */

package silos;

/**
 * Esta clase permite crear objetos de tipo SiloEsfera3
 *
 * @author mdomitsu
 */
public class SiloEsfera3 extends Silo3 implements ISilo {
  private double radio;
  private int tipoBase;

  /**
   * Construye un objeto de esta clase e inicializa sus atributos
   * @param radio Radio del silo esférico
   */
  public SiloEsfera3(double radio, int tipoBase, int grosorLamina) {
    // Llama al constructor de la clase padre: Silo3
    super("Silo Esférico", grosorLamina);

    // Inicializa los atributos
    this.radio = radio;
    this.tipoBase = tipoBase;
  }

  /**
   * Calcula la superficie de la esfera
   */
  public void calculaSuperficie() {
    superficie = 4*Math.PI*radio*radio;
  }

  /**
   * Calcula el volumen de la esfera
   */
```

```
  public void calculaVolumen() {
    volumen = 4*Math.PI*radio*radio*radio/3;
  }

  /**
   * Calcula el precio del silo esférico
    */
  public void calculaCosto() {
    costo = Costos.get(tipoBase) + superficie *
            Costos.get(grosorLamina);
  }

  /**
   * Genera una cadena con la representación de esta clase
   * @return Una cadena con la representación de esta clase
   */
  public String toString() {
    return super.toString() + ", radio = " + radio + ", superficie = "
           + superficie + ", volumen = " + volumen + ", costo: "
           + costo;
  }
}
```

Para probar las clases anteriores se implementa la siguiente clase de prueba:

PruebaSilo3

```
/*
 * PruebaSilo3.java
 *
 * Creada el 7 de octubre de 2005, 12:45 PM
 */

package silos;

/**
 * Esta clase se utiliza para probar las clases SiloCilindro3,
 * SiloCono3 y SiloEsfera3
 *
 * @author mdomitsu
 */
public class PruebaSilo3 {

  /**
   * Método main() en el que se invocan a los métodos de las clases
   * SiloCilindro3, SiloCono3 y SiloEsfera3 para probarlos
   * @param argumentos Los argumentos en la línea de comando
   */
  public static void main(String[] args) {
    PruebaSilo3 pruebaSilo3 = new PruebaSilo3();

    ISilo silo[] = new ISilo[5];

    // Crea 5 silos de diferente tipo
    silo[0] = new SiloCilindro3(1.0, 1.0, Costos.LAMINA_CAL12);
    silo[1] = new SiloEsfera3(1.0, Costos.BASE_SIMPLE,
                              Costos.LAMINA_CAL12);
    silo[2] = new SiloCono3(1.0, 1.0, Costos.BASE_SIMPLE,
                            Costos.LAMINA_CAL12);
    silo[3] = new SiloCilindro3(2.0, 1.0, Costos.LAMINA_CAL14);
    silo[4] = new SiloCilindro3(1.0, 2.0, Costos.LAMINA_CAL16);
```

```
    // Para cada silo
    for (int i = 0; i < 5; i++) {
        // Calcula la superficie del silo
        silo[i].calculaSuperficie();

        // Calcula el volumen del silo
        silo[i].calculaVolumen();

        // Calcula el costo del silo
        silo[i].calculaCosto();
    }

    // Para cada silo
    for (int i = 0; i < 5; i++)
      // Escribe los valores de sus atributos
      System.out.println(silo[i]);

    // Escribe el número de silos creados
    System.out.println("Silos creados: " + Silo3.contadorSilos);
  }
}
```