Capítulo

4

Programación Orientada a Objetos

El capítulo asume que el lector posee conocimientos básicos de Programación Orientada a Objetos o POO en C++, así como conocimientos de la estructura general y funcionamiento de programas en Java y C#. Los temas tratados refuerzan dichos conocimientos básicos y profundizan en una amplia variedad de conceptos avanzados. Estos conceptos son vistos bajo la implementación de los lenguajes de programación C++, Java y C#, con el objetivo de reforzar dichos conceptos más allá de la implementación particular de un lenguaje, mediante la identificación de las diferencias y similitudes entre éstos.

Conceptos Básicos

En esta sección se busca revisar los conceptos básicos relacionados a las clases y a los objetos, así como a los miembros que los componen.

Sobre las Clases

Uno de los primeros progresos de abstracción en el manejo de tipos de datos fue la especificación de los llamados Abstract Data Types o ADT. Un ADT es un conjunto de valores de datos y operaciones asociados a éstos, los que son especificados de manera precisa, independientemente de alguna implementación en particular. Los ADT fueron llevados a la práctica en lenguajes como Clu, Mesa y C. Por ejemplo, los ADT en C se implementaron como estructuras y un conjunto de funciones capaces de manipularlas. Siempre que el acceso a la composición interna de estas estructuras sea realizado por dicho conjunto especializado de funciones, se oculta dicha composición.

A este concepto se le conoce como encapsulación. Este concepto buscaba los siguientes objetivos:

- Reducir el efecto producido en un sistema por la modificación de una estructura de datos, facilitando este tipo de labor.
- Focalizar los cambios en las estructuras de datos y en sus operaciones relacionadas, siempre que ambas se puedan circunscribir a la misma ubicación en el código fuente.
- Facilitar la localización y corrección de errores, reduciendo además el alcance de éstos.

La POO va más allá, haciendo que dichas funciones de manipulación formen parte de la descripción del tipo de dato. Juntos, la declaración de datos y las funciones de manipulación, forman el concepto de clase, donde un objeto es una instancia de ésta. La POO también agregó los conceptos de herencia de implementación y comportamiento polimórfico. Los conceptos de POO fueron llevados a la práctica por lenguajes como Ada, C++ y SmallTalk. Es importante señalar que existen diferentes opiniones

acerca de los conceptos relacionados con los ADT y las clases, algunos autores incluso los intercambian indistintamente.

Dado que la encapsulación realiza el ocultamiento de información de un objeto, debido a la inseguridad de su manipulación libre y directa, se hace necesario definir un mecanismo que permita realizarla de manera segura. Es aquí donde se define el concepto de interfaz de una clase, la cual es el mecanismo que dicha clase define para comunicarse con su entorno. En la práctica, la interfaz de una clase está formada principalmente por un conjunto de funciones, que forman parte de la definición de la clase, a las que se puede acceder y llamar desde fuera de la clase.

Sobre los Objetos

Las clases son un mecanismo de definición de nuevos tipos de datos. Un objeto es una instancia de una clase. Dentro del contexto de la POO, el concepto de clase corresponde al de tipo de dato, mientras que el concepto de objeto corresponde al de objeto de datos (ver Capítulo 2 la definición de "objeto de datos").

Al igual que los tipos predefinidos de un lenguaje de programación, los nuevos tipos de datos creados utilizando clases deben implementarse de manera que sus objetos siempre estén en un estado consistente. Por ejemplo, sería inconsistente que la suma de dos enteros provoque inadvertidamente la modificación de los sumandos. Debido a que el estado de un objeto corresponde al de sus datos miembros, la implementación de los métodos de una clase deberá realizarse de manera que cuide que dichos datos se mantengan en un estado consistente. Esta consistencia debe asegurarse durante todo el tiempo de vida de cada objeto de la clase, desde su construcción hasta su destrucción. Para lograrlo, los lenguajes orientados a objetos permiten definir "constructores" y "destructores" en las clases.

Sobre los Miembros

Los miembros son los elementos que pueden declararse dentro de una clase y forman parte de ella.. Las clases pueden tener 3 tipos de miembros:

- **Datos:** Llamados datos miembro o atributos. El término atributo es utilizado para un concepto diferente en ..NET; por lo tanto, no se utilizará en este documento.
- **Funciones:** Llamadas métodos.
- **Tipos de datos:** Llamados tipos anidados.

Cada miembro de una clase tiene un modificador de acceso relacionado ya sea implícita o explícitamente. En el caso implícito se aplica el modificador de acceso por defecto correspondiente al lenguaje.

Las Clases

En esta sección veremos las diferencias y similitudes entre la implementación de los lenguajes de programación tratados respecto a la POO.

C++ es una extensión de un lenguaje de programación estructurado, por lo que muchos se refieren a éste como "un lenguaje estructurado con características orientadas a objetos", más que "un lenguaje orientado a objetos real". Todo programa en C++ contiene por lo menos una función fuera de las clases que define, la función de entrada "main", por lo que ningún programa en C++ es completamente orientado a objetos.

A diferencia de C++, los lenguajes orientados a objetos Java y C# son más estrictos respecto a su implementación del paradigma orientado a objetos. En estos lenguajes, todo programa consiste en un conjunto de clases, donde una de dichas clases debe definir un método que será tratado por el intérprete como el punto de entrada del programa. A una clase que define un punto de entrada de un programa la llamaremos la "clase ejecutable".

En Java y C# toda clase debe contener también su implementación completa. Java y C#, a diferencia de C++, no permiten la declaración de prototipos de métodos dentro de una clase para su posterior implementación fuera de ésta. No existe por tanto, en estos lenguajes, un equivalente a la declaración de una clase en un archivo cabecera (*.H) y su implementación en un archivo de implementación (*.CPP). En Java y C# una clase se declara e implementa en un mismo archivo. Tampoco permiten la declaración de variables globales ni funciones fuera de las clases.

La Declaración de las Clases

El formato general de declaración de una clase en Java es:

```
[modificadores] class <nombre> [extends <clase base>] [implements <lista interf.>]
{
   <definición de campos y métodos>
}
```

El formato general de declaración de una clase en C# es:

```
[modificadores] class <nombre > [ : <clase base y/o lista de interfaces> ]
{
   <definición de campos y métodos>
}
```

A diferencia de C++, Java y C# permiten utilizar modificadores en la declaración de las clases. La tabla 4.1 muestra los modificadores que pueden utilizarse.

Tabla 4.1. Modificadores de clases para Java y C#

Modificadores de Java	Modificadores de C#	Descripción
abstract	abstract	Permiten que una clase no sea instanciada.
final	sealed	Evitan que una clase sea utilizada como base para otra clase.
public de-paquete (*)	public protected private internal (*)	Determinan el acceso a la clase (por ejemplo, para crear objetos de ésta) desde fuera de su ámbito de declaración. Dependiendo del contexto en que la clase se declare, alguno de estos modificadores no pueden utilizarse. (*) Modificador por defecto.

La especificación de la herencia se hace mediante la palabra reservada extends en Java, y con el símbolo ":" en el caso de C#. Cuando una clase no hereda explícitamente de otra, Java asume que ésta hereda de la clase Object, mientras que C# asume que lo hace de la clase object. En este sentido, toda clase en Java y C# hereda, directa o indirectamente, de una clase base común. A este esquema de implementación de clases se le llama de "árbol único", dado que toda clase tiene una raíz común. La implementación de C++ se considera de "árbol múltiple". La implementación de árbol único permite darle una funcionalidad básica común a todos los objetos creados por un programa, sacrificando algo de la eficiencia que se puede lograr con la implementación de árbol múltiple.

Es importante notar que los privilegios de acceso, a diferencia de C++, se definen al declararse la clase, no al usarla como base de una herencia. En el sentido de C++, se puede considerar que todas las herencias en Java y C# son públicas. Se verá el tema de la herencia con detalle más adelante.

La Creación de los Objetos

Los objetos son las instancias de las clases. En C++, Java y C# la creación de un objeto utiliza la palabra reservada "new" bajo una sintaxis similar.

El siguiente programa define y utiliza una clase Usuario en Java.

```
class Usuario {
    public static final int Administrador = 0;
    public static final int Registrador = 1;
    public static final int Verificador = 2;
    public static final int Consultor = 3;
    private String nombre;
    private String contraseña;
    private int tipo;

    public Usuario(String nom, int tipo) {
        nombre = nom;
        contraseña = "";
        this.tipo = tipo;
    }
    public String ObtenerNombre() {
        return nombre;
    }
}
class Principal {
    public static void main(String[] args) {
        Usuario p;
        p = new Usuario("Jose", Usuario.Administrador);
        System.out.println("Buenos dias Sr(a).  " + p.ObtenerNombre());
    }
}
```

El siguiente programa es el equivalente en C#.

```
using System;
enum TipoUsuario {
    Administrador,
    Registrador,
    Verificador,
    Consultor
}
class Usuario {
    private string nombre;
    private TipoUsuario tipo;
    public Usuario(string nom, TipoUsuario tipo) {
        nombre = nom;
        this.tipo = tipo;
    }
    public string ObtenerNombre() {
        return nombre;
    }
}
class Principal {
    public static void Main(string[] args) {
        Usuario p;
        p = new Usuario("Jose", TipoUsuario.Administrador);
        Console.WriteLine("Buenos dias Sr(a).  " + p.ObtenerNombre());
    }
}
```

Si bien la sintaxis es muy similar a C++, existen algunas diferencias importantes:

- Todos los modificadores de acceso, public y private, en el ejemplo, deben especificarse independientemente por cada miembro de la clase, sea un método o un campo. Si no se especifica, se asume un modificador de acceso por defecto (ver tabla 4.2).

- La definición de una clase no termina en un ';'.

- La creación de objetos se realiza con el operador new, pero no existe un operador delete, dado que los intérpretes de Java y C# se encargan de gestionar la liberación de la memoria.

- El acceso a los miembros de una clase u objeto siempre es con el operador punto. En Java y C# los objetos son reservados automáticamente en el montón (en pila en algunos casos en C#, como se verá más adelante). el programa no puede decidir esto como sí sucede en C++.

- Todo el código del programa debe estar dentro de los métodos de las clases.

Entre Java y C# también existen diferencias importantes:

- No existen enumerados en Java, por lo que el ejemplo anterior utilizó constantes. Existen otros tipos de datos propios de C#, sin equivalente en Java e incluso en C++.

- El uso de librerías difiere en Java y C#. Para el programa en C# se utilizó la palabra reservada "using", dado que C# no importa por defecto ninguna librería, ni siquiera las básicas. Para el programa en Java no se requirió realizar un "import", debido a que solo se utilizó las librerías básicas, las que son importadas por defecto.

Existen otras diferencias que se irán viendo a lo largo del presente capítulo.

Los Constructores y Destructores

El concepto de encapsulación de la POO tiene como objetivo que los programas mantengan sus objetos en un estado consistente durante todo su ciclo de vida. Los constructores son métodos llamados durante la creación de los objetos de forma que éstos tengan un estado consistente desde su nacimiento. Los destructores aseguran una correcta liberación de los recursos reservados por el objeto antes de que éste sea eliminado.

Los Constructores

Las declaraciones de los constructores en C++, Java y C# son:

C++:

```
class MiClase {
      public:
      MiClase( ....  ) { ....  }
};
```

Java:

```
class MiClase {
      public MiClase( ....  ) { ....  }
}
```

C#:

```
class MiClase {
      public MiClase( .......  ) { .......  }
}
```

Los constructores pueden recibir un conjunto de parámetros, igual que cualquier método, para permitir establecer un estado inicial consistente. Cuando un constructor no recibe parámetros se le llama "constructor por defecto", cuando su parámetro es un objeto del mismo tipo se le llama "constructor copia". Dado que la implementación interna de muchos lenguajes de programación orientados a objetos requiere que siempre se llame a un constructor cuando se crea un objeto, cuando no se implementa uno explícitamente (sea éste uno "por defecto" o no), el compilador del lenguaje crea un constructor por defecto implícitamente.

C++ implementa una llamada automática al "constructor copia" de una clase cuando se crea un objeto en pila y se le inicializa, pasándole otro objeto del mismo tipo, en su misma declaración. Este manejo especial de los constructores copia no es implementado ni en Java ni en C#, ni tampoco permiten la sobrecarga del operador de asignación para estos fines, lo que sí permite C++.

Java y C# permiten llamar a un constructor desde otro constructor de la misma clase. Esto permite simplificar la definición de varios constructores opcionales para la creación de un objeto. El siguiente código muestra esta característica en C#.

```
class Usuario {
    private string nombre;
    private string contraseña;
    public Usuario(string nombre) : this(nombre, nombre) {
        // este constructor llama a su vez al segundo constructor pasándole como
        // contraseña el mismo nombre
    }
    public Usuario(string nombre, string contraseña) {
        this.nombre = nombre;
        this.contraseña = contraseña;
    }
}
```

El siguiente código es el equivalente en Java:

```
class Usuario {
    private String nombre;
    private String contraseña;
    public Usuario(String nombre) {
        this(nombre, nombre)
    }
    public Usuario(String nombre, String contraseña) {
        this.nombre = nombre;
        this.contraseña = contraseña;
    }
}
```

Aunque la sintaxis utilizada en C# corresponde a la lista de inicialización de los constructores en C++, no es posible utilizar esta sintaxis para inicializar datos miembros de la clase, como sí ocurre en C++.

Como se verá más adelante, en Java y C# también es posible llamar desde un constructor a otro constructor de la clase base.

Los constructores, como cualquier miembro de una clase, pueden también recibir modificadores de acceso en su declaración. Los modificadores de acceso que pueden utilizarse dependen en algunos casos de si la clase es anidada o no. Las clases anidadas se verán más adelante.

Los Constructores Estáticos

A los constructores vistos en la sección anterior se les llama también "de instancia", debido a que se llaman cuando se instancia una clase. Su finalidad es la de establecer el estado inicial consistente de un nuevo objeto. Sin embargo, una clase puede requerir inicializar datos cuyo tiempo de vida abarquen a todos los objetos de la clase, es decir, datos miembros estáticos. Para este tipo de inicialización se definen constructores estáticos. Java y C# permiten este tipo de constructores. Ejemplos de su declaración en Java y C# son:

Java:

```
class MiClase {
    static { .......  }
}
```

C#:

```
class MiClase {
    static MiClase() { .......  }
}
```

El siguiente programa en C# muestra el uso de un constructor estático.

```
class NumeroAleatorio {
    private static int semilla;
    static NumeroAleatorio() {
        // imaginemos que obtenemos un valor semilla de,
        // por ejemplo, el reloj del sistema
        semilla = 100;
    }
    public static int sgte() {
        // imaginemos que generamos un número aleatorio
        return semilla++;
    }
}
class Principal {
    public static void Main(string[] args) {
        Console.WriteLine("aleatorio 1 = " + NumeroAleatorio.sgte());
        Console.WriteLine("aleatorio 2 = " + NumeroAleatorio.sgte());
    }
}
```

Nótese que el constructor estático no define un nivel de acceso ni tampoco puede tener parámetros, debido a que nunca es llamado por otro código del programa, sólo por el intérprete de Java o ..NET cuando la clase es cargada. C++ no tiene un equivalente a un constructor estático.

Los Destructores

Mientras que la construcción de un objeto sigue esquemas similares de funcionamiento en la mayoría de lenguajes orientados a objetos , la destrucción de los mismos depende de si el esquema de manejo de la memoria es determinista o no-determinista, lo que está relacionado con el momento en que un destructor es llamado. El concepto mismo de destructor de C++ es determinista, lo que significa que el programador puede determinar el momento exacto en que un destructor es llamado durante la ejecución de un programa. Java y C# implementan “finalizadores”, los cuales son no-deterministas.

La sintaxis de declaración de un finalizador en Java es:

```
<modificador protected o public> finalize ( ) {
    < Cuerpo del finalizador>
}
```

La sintaxis de declaración de un finalizador en C# es:.

```
~< nombre de la clase > ( ) {
    < Cuerpo del finalizador>
}
```

En ambos casos no se pueden definir parámetros. Tanto en C++ como en C# no es posible llamar a un destructor o finalizador directamente desde fuera de la clase. En el caso de Java el finalizador es un método más pero con un nombre reservado, por lo que es posible llamarlo directamente desde fuera de la clase. Más aún, este método sobrescribe el método de la clase base dado que la clase Object lo implementa. Las consecuencias de esto se verán en el tema de la herencia.

Debido a que el manejo de la memoria, el recurso más común de los programas, es realizado por el recolector de basura en Java y C#, los finalizadores son rara vez requeridos y se recomienda utilizarlos sólo cuando es estrictamente necesario (dado que su gestión disminuye la performance de un programa), para liberar recursos que no corresponden a la memoria (como por ejemplo, una sesión abierta en una base de datos, el acceso a un puerto del computador, entre otros), o para realizar un seguimiento o depuración de un programa.

La Recolección de Basura

Si bien un objeto se crea utilizando el operador "new", no existe en Java y C# un operador "delete". El motivo de esto es que Java y C# le quitan al desarrollador la responsabilidad de la liberación de los recursos creados dinámicamente. Los intérpretes de estos lenguajes guardan un registro de todos los objetos creados y cuántas variables de tipo referencia apuntan a cada uno. Cuando una nueva referencia apunta a un objeto, el contador de referencias de éste aumenta. Cuando una referencia a un objeto se destruye (por ejemplo, una referencia local, la cual se destruye cuando se sale del método que la define) el contador de referencias del objeto relacionado disminuye. De esta forma, cuando el contador de un objeto llega a cero se sabe que ya no es referenciado por ninguna variable del programa. Paralelamente, estos intérpretes cuentan con un sistema llamado recolector de basura, el cual es llamado cada cierto tiempo de forma automática y verifica todos los objetos que ya no cuentan con referencias, esto es, su contador de referencias es cero. Cuando el recolector encuentra un objeto sin referencias lo elimina de memoria. De esta forma Java y C# simplifican enormemente el trabajo con la memoria dinámica.

Manejo Manual y Automático de la Memoria

C++ otorga al programador toda la responsabilidad del manejo de la memoria, lo cual le permite desarrollar programas significativamente más eficientes que sus equivalentes en Java o C#. Sin embargo, la experiencia ha demostrado que esta responsabilidad ha generado más sobrecostos que beneficios para la mayoría de proyectos de software. Estos sobrecostos se traducen en mayores tiempos de desarrollo y depuración de errores durante las etapas de desarrollo y mantenimiento de los programas, así como en posteriores proyectos que involucran la modificación y ampliación de éstos.

La automatización de la gestión de la memoria, ofrecida por Java y C#, quita esta responsabilidad al programador, permitiéndole concentrar su trabajo en la implementación de la llamada "lógica del negocio", lo que ha permitido generar programas más estables en menor tiempo, con un menor número de errores durante y después de su producción, y facilitando la posterior modificación de dichos programas. Todo esto, a costa de una pérdida "aceptable" de la performance final de estos programas respecto a sus contrapartes. Más aún, se ha encontrado que en muchos casos un gestor automático de memoria puede tomar decisiones más eficientes en cuanto al manejo de memoria que un programador promedio. Como consecuencia, para proyectos de mediana y gran envergadura, las empresas requerirían contar con programadores expertos en C++ para lograr resultados "significativamente" más eficientes que los obtenidos con programadores "no-expertos" utilizando Java y C#.

Sin embargo, existen indudablemente proyectos cuya naturaleza requiere obtener la mayor eficiencia que el hardware y el software disponible puede brindar, por lo que lenguajes como Java y C# no pueden reemplazar completamente a lenguajes como C++. Como en muchos aspectos del desarrollo de software, el escoger un lenguaje apropiado para el desarrollo de un programa, pasa por conocer cuáles son los requerimientos del producto final.

Finalización Determinista y No-Determinista

El manejo automático de la memoria ofrecido por Java y C# implica que dichos lenguajes faciliten el desarrollo de programas no-deterministas. En un esquema no-determinista el programador no puede determinar en qué momento los objetos que crea, luego de que ya no son referenciados por el programa, son efectivamente eliminados de la memoria. Esto trae consecuencias al momento de decidir la forma en que los recursos reservados por un programa deberán ser eliminados.

Para esto es importante considerar que si bien la memoria es el recurso más comúnmente manejado por los programas, no es el único, por lo que el resto de recursos deberán seguir manejándose de una manera determinista.

Por ejemplo, el sistema operativo Windows permite a los programas crear un número limitado de "manejadores de dispositivos de las ventanas". Debido a esto, una librería que permita crear ventanas en Java y C# deberá controlar en forma determinista la liberación de estos recursos. Un programa que

utilice dicha librería puede crear y luego desechar una gran cantidad de ventanas en un corto período de tiempo. Si el diseñador de esta librería coloca en el finalizador de su clase "Ventana" la llamada al sistema operativo que libera el recurso mencionado, es posible que las llamadas a estos finalizadores tarden lo suficiente como para que el programa trate de crear una nueva ventana y se produzca un error, puesto que ya se crearon todas las que el sistema operativo permitía y su liberación aún está pendiente.

Problemas como el descrito requieren que el programador implemente manualmente un manejo determinista de estos recursos, es decir, se requiere tener la certeza de en qué momento se libera un determinado recurso.

Los recolectores de basura de Java y C# pueden ser controlados hasta cierto punto por el programador, haciendo uso de clases especiales. Estos recolectores ofrecen métodos para despertar manualmente el procedimiento de recolección de basura. Sin embargo, esta solución no es confiable, puesto que el recolector de basura siempre tiene la potestad de decidir si es conveniente o no iniciar la recolección. El no hacerlo así, podría ocasionar que el programa se cuelgue debido a, por ejemplo, un finalizador que es ejecutado por el recolector y que nunca finaliza por un error en la lógica de su código, lo que ocasionaría que el programa se quede esperando a que el proceso de recolección termine, lo que nunca ocurrirá.

Un esquema comunmente utilizado pasa por colocar el código encargado de la liberación de la memoria en un método del objeto que la crea. Adicionalmente, el objeto deberá contener un dato miembro que funcione como bandera y que indique si dicho método ya fue llamado o no. Si el programador determina que ya no requiere utilizar más un objeto, y sus recursos, deberá llamar manualmente a dicho método. Adicionalmente, el finalizador del objeto también llamará a este método. Finalmente, todos los métodos del objeto deberán hacer una verificación de la bandera de liberación de los recursos antes de realizar su trabajo, de forma que se impida que el objeto siga siendo utilizado si es que sus recursos ya fueron liberados.

Puede parecer que finalmente la programación no-determinista no es tan buena idea después de todo. Sin embargo, la mayor parte de los recursos que un programa suele utilizar corresponden a la memoria, y para aquellos recursos que no son memoria suelen contarse con librerías especializadas que hacen casi todo el trabajo no-determinista por nosotros.

El Acceso a los Miembros

El acceso a los miembros de una clase desde fuera de la misma se determina mediante los modificadores de acceso de cada miembro. La tabla 4.2 muestra los modificadores de acceso de cada lenguaje según los ámbitos desde dónde se desea que un miembro sea accesible.

Tabla 4.2. Modificadores de acceso para miembros de una clase

Ámbitos	C++	Java	C#
Sólo desde dentro de la clase donde es declarada.	private (*)	private	private (*)
Desde la clase donde es declarada y las que deriven de ella, estén o no en la misma librería.	protected	*<no definido>*	protected
Desde cualquier clase de la misma librería.	*<no definido>*	de-paquete (*)	internal
Desde cualquier clase de la misma librería y desde otras clases en otras librerías siempre que deriven desde la clase donde es declarada.	*<no definido>*	protected	protected internal
Desde cualquier clase en cualquier librería.	public	public	public

(*) Modificador por defecto.

De la tabla se puede ver que los modificadores de acceso están relacionados con dos conceptos: La herencia y las librerías. También se puede ver que Java modifica el concepto original de C++ de lo que es un miembro protegido. El siguiente ejemplo en Java muestra los efectos del modificador protected en Java.

```
// Archivo Alpha.java
```

```
package PaqueteX;
public class Alpha {
    protected int protectedData;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}

// Archivo Gamma.java
package PaqueteX;
public class Gamma {
    public void accessMethod(Alpha a) {
        a.protectedData = 10;    // legal
        a.protectedMethod();    // legal
    }
}

// Archivo Delta.java
package PaqueteY;
import PaqueteX.Alpha;
public class Delta extends Alpha {
    public void accessMethod(Alpha a) {
        a.protectedData = 10;    // ilegal
        a.protectedMethod();    // ilegal
        protectedData = 10;    // legal
        protectedMethod();    // legal
    }
}

// Archivo PruebaDeProtected.java
import PaqueteX.Alpha;
import PaqueteX.Gamma;
import PaqueteY.Delta;
public class PruebaDeProtected {
    public static void main(String[] args) {
       Alpha a = new Alpha();
       Gamma g = new Gamma();
       Delta d = new Delta();
       a.protectedData = 10;    // ilegal
       a.protectedMethod();    // ilegal
       g.accessMethod(a);
       d.accessMethod(a);
    }
}
```

En el caso de la clase Gamma es posible acceder a los miembros protegidos de Alpha, puesto que ambas clases pertenecen al mismo paquete. La clase Delta no puede acceder a los miembros protegidos de Alpha dado que no pertenecen al mismo paquete, excepto sus propios miembros heredados. La clase PruebaDeProtected no puede acceder a los miembros protegidos puesto que no pertenece al mismo paquete.

El Uso de las Variables

Los datos de un programa son manejados mediante variables. La forma y ámbito en que éstas son declaradas determinan muchas de sus características.

La Declaración de las Variables

En Java y C# la declaración de variables en Java es similar a C++. La sintaxis general de declaración de una variable es:

```
[modificadores] <tipo> <nombre> [ = <inicializador>];
```

En Java, las variables se clasifican en:

- Variables de tipo primitivo, correspondientes a los tipos byte, short, int, long, float, double, char y boolean.

- Variables de tipo objeto, creadas sobre la base de tipos de dato “clase”.

En C#, las variables se clasifican en:

- Variables por valor, correspondientes a los tipos primitivos y a las estructuras. Son variables que se reservan en pila.

- Variables por referencia, correspondientes a las clases, interfaces, entre otros. Son variables que representan una referencia en pila a datos reservados en el montón.

Los tipos primitivos no son objetos en Java, lo que permite que su manejo sea más eficiente en cuanto a la memoria utilizada para su almacenamiento y el tiempo de acceso a sus datos. En C# los datos primitivos sí son objetos, lo que le permite a un programa manejar cualquier objeto de datos como un objeto, sacrificando un poco de eficiencia. En este sentido, C# es más estrictamente orientado a objetos que Java. La librería estándar de Java ofrece además clases que encapsulan a cada uno de sus datos primitivos, para aquellos programas que requieren manipular todos sus datos como objetos.

Por el ámbito donde son declaradas, las variables se clasifican en:

- Variables de clase, que corresponden a los datos miembros de la clase.

- Variables locales, que corresponden a las variables declaradas dentro de los métodos.

C++ permite adicionalmente la declaración de variables globales, lo que no permiten ni Java ni C#.

La Duración y el Ámbito

Todo objeto de datos relacionado a una variable en C++, Java y C#, posee dos características que determinan cómo puede ser utilizado: Su duración y su ámbito.

La duración determina el período en el que un objeto de datos existe en memoria y por tanto, su variable puede ser utilizada. Existen 3 casos:

- Creados y destruidos automáticamente. Corresponde a los datos locales a los métodos (declarados dentro de éstos). Se suele decir que tiene una duración automática, dado que son creados automáticamente cuando el programa entra en el bloque en que son declarados y se destruyen automáticamente al salir de éste. Sus variables son llamadas “variables automáticas”.

- Creados y destruidos conjuntamente con los objetos de los que son miembros. Su duración está ligada a la duración del objeto creado. Sus variables son llamadas “variables de instancia”.

- Creados una sola vez y que existen durante toda la ejecución del programa. Corresponden a los datos miembro estáticos de una clase. Corresponden a las “variables estáticas” y “variables globales” para el caso de C++.

El ámbito determina desde dónde un dato puede ser referido dentro de un programa. Existen 3 tipos de ámbito:

- Ámbito de Clase: Son los datos miembros de una clase. Pueden ser referidos desde cualquier método dentro de dicha clase y desde métodos fuera de ésta siempre que los modificadores de ámbito aplicados lo permitan. Se verá más adelante cómo se aplican dichos modificadores.

- Ámbito de Bloque: Son datos definidos dentro de un bloque, como por ejemplo el bloque que representa el cuerpo de un método o el bloque de una estructura de control (if, for, while y do / while). Estos datos sólo pueden ser referidos dentro del bloque donde son declarados.

✍ Ámbito Global: Son los datos correspondientes a variables declaradas fuera de las clases y métodos. Sólo soportado en C++.

Un dato en ámbito de bloque oculta otro, con el mismo nombre, en el ámbito de la clase a la que pertenece. El siguiente programa en Java muestra un ejemplo de esto:

```
class PruebaDeAmbito
{
  int iValor = 10;
  …

  void Prueba()
  {
    int iValor = 20;
    System.out.println( “iValor = ” + iValor );
    System.out.println( “this.iValor = ” + this.iValor );
  }
  …

}
```

Al ejecutar el método Prueba se producirá la siguiente salida:

```
iValor = 20
this.iValor = 10
```

La variable local iValor oculta a la variable de la clase. Para poder hacer referencia a la variable miembro de la clase usamos la palabra reservada this, de la misma forma que se realiza en C++. this representa la referencia de un objeto a sí mismo dentro de uno de sus métodos.

Dentro de un método no se pueden declarar dos variables con el mismo nombre, aún si están declaradas en bloques distintos. El siguiente programa en Java muestra un ejemplo de esto:

```
void Prueba()
{
    int iValor = 20;
    boolean Flag = true;
    if( Flag )
    {
      int iValor;
      …
    }
}
```

Al ejecutarse se producirá un error de compilación. El siguiente ejemplo muestra esta misma restricción en C#:

```
class PruebaDeAmbito {
   public int entero;
   public void prueba(int entero) {
       this.entero += entero;
       if(entero < 0) {
          int entero = 0; // ERROR
          double real = 2.5;
       } else {
          double real = 3.8;
       }
   }
}
```

En el código, definir un argumento o variable local con el mismo nombre de una variable de clase oculta ésta última, por lo que se requiere utilizar la palabra reservada this, como en C++. Sin embargo, la definición de la variable **“entero”** dentro del cuerpo de la estructura de control if produce un error de compilación, dado que ya existe una variable local con el mismo nombre en un ámbito padre, el del

método. Esto difiere al caso de las variables **"real"**, cuya declaración no produce un error dado que el ámbito de una no es padre del otro.

La Inicialización Automática

A diferencia de C++, en Java y C# toda variable puede ser inicializada al momento en su declaración, inclusive los datos miembros, con excepción de los parámetros de un método, que son también variables locales, dado que su inicialización corresponde a los valores pasados al momento de llamarse a dicho método. El siguiente código en C# muestra esta inicialización:

```
class Usuario {
    private string nombre = "Jose";
    private int edad = 10;
}
```

Las variables locales deben inicializarse antes de ser utilizadas. Por ejemplo, el siguiente código en Java arrojará un error al momento de compilarse:

```
String sTexto;
sTexto += "Hola ";
sTexto += "mundo";
```

Este código puede corregirse de la siguiente forma:

```
String sTexto;
sTexto = "Hola ";
sTexto += "mundo";
```

De la forma:

```
String sTexto = new String( );
sTexto += "Hola ";
sTexto += "mundo";
```

O bien de la forma:

```
String sTexto = new String( "Hola " );
sTexto += "mundo";
```

A diferencia de las variables locales, cuando una variable de clase no es inicializada por el programa durante la creación del objeto al que pertenece, recibe una inicialización por defecto. Esta inicialización se realiza de la siguiente forma:

- Los datos primitivos numéricos se inicializan en cero.
- Los datos primitivos lógicos (boolean en Java, bool en C#) se inicializa en false.
- Las referencias se inicializan en null.

Los Modificadores

Dentro de un método, los únicos modificadores permitidos para las variables son "final" en Java, para variables cuyo valor sólo puede asignarse una vez, y "const" en C#, para constantes.

La siguiente tabla muestra los modificadores permitidos, fuera de los modificadores de acceso, en Java y C# para variables declaradas en el ámbito de una clase.

Tabla 4.3. Modificadores de Variables en Java y C#

C#	Java	Descripción
Static	static	Variable estática.

Const	final static	Constante.
<sin equivalente>	final	Variable cuyo valor sólo puede asignarse una vez, en la declaración o después.
Readonly	<sin equivalente>	Variable cuyo valor sólo puede ser asignado durante la creación del objeto.
<sin equivalente>	transient	Variable no serializada con el objeto durante un proceso de serialización. (*)
<sin equivalente>	volatil	Variable para que el compilador no realice ciertas optimizaciones al momento de generar el BYTECODE que hará uso de él. (*)
New		Variable que oculta otra variable, con el mismo nombre, en una clase base.

(*) El uso de estos modificadores va más allá del alcance del presente curso.

El acceso a los datos miembros estáticos en C# se diferencia a C++ y Java en que:

- No pueden accederse, desde dentro de la clase, mediante la palabra reservada this.
- No pueden accederse, desde fuera de la clase, mediante una referencia, debe hacerse mediante el nombre de la clase.

El siguiente código en C# presenta estos dos casos.

```
class Usuario {
    ...
    public static int cuenta = 0;
    public Usuario(string nom, TipoUsuario tipo) {
        this.cuenta++;// ERROR
        cuenta++;//CORRECCIÓN

    ...
    }
    ...
}
class Principal {
    public static void Main(string[] args) {
        Usuario p;
        p = new Usuario("Jose", TipoUsuario.Administrador);

        Console.WriteLine("Número de usuarios = " + p.cuenta); // ERROR
        Console.WriteLine("Número de usuarios = " + Usuario.cuenta); // Corrección

    }
}
```

La palabra reservada readonly define un dato miembro de sólo lectura, el que sólo puede recibir un valor durante el proceso de creación del objeto, esto es, en la misma declaración de la variable o dentro de un constructor. El siguiente código muestra un ejemplo de este tipo de variable.

```
using System;
class PruebaDeSoloLectura {
    private readonly int valor = 50;
    public PruebaDeSoloLectura() {
        valor = 100;
        valor += 10;
    }
    public void modificar(int nuevo) {
        valor = nuevo;   // ERROR
    }
}
class Principal {
    public static void Main(string[] args) {
        PruebaDeSoloLectura obj = new PruebaDeSoloLectura();
        obj.modificar(200);
    }
}
```

En el código anterior, el dato miembro **"valor"** se inicializa al declararse y su valor es modificado dos veces dentro del constructor. Sin embargo, la modificación de dicho dato miembro en el método modificar produce un error en tiempo de compilación.

La palabra reservada const tiene el mismo significado en C# y C++. Sin embargo, una constante en C# es implícitamente estática, mientras que en C++ no. En C#, los datos miembro const se diferencian de los readonly en que:

- Se pueden definir constantes tanto de clase como locales.
- Deben de ser inicializadas en su declaración.
- Su valor debe poder ser calculado en tiempo de compilación.
- Son implícitamente static.

La palabra reservada new se verá como parte del tema del manejo de la herencia.

El Uso de los Métodos

Mientras que en C++ pueden definirse funciones tanto dentro como fuera de las clases, en Java y C# sólo pueden definirse funciones dentro de las clases. A las funciones definidas dentro de una clase se les denomina funciones miembro o métodos.

La Declaración de los Métodos

En Java, el formato general de definición de un método es:

```
[modificadores] <tipo-retorno> <nombre> ([lista de parámetros])
[throws <lista de excepciones>]
{
   <cuerpo del método>
}
```

En C#, el formato general de definición de un método es:

```
[modificadores] <tipo-retorno> <nombre> ( [lista de parámetros] )
{
   <cuerpo del método>
}
```

En Java y C#, tanto la lista de parámetros como los modificadores son opcionales. El valor de retorno puede ser cualquier tipo de dato o bien puede ser void, lo que indica que el método no devuelve ningún valor.

En Java y C#, un método sin parámetros no puede llevar la palabra void entre los paréntesis, como si se puede hacer en C++. A diferencia de C++, no se pueden declarar prototipos de métodos para definirlos fuera de la clase. Un método sólo puede ser definido dentro de una clase.

Un método no puede definirse dentro de otro método, ni fuera de una clase, lo que sí puede hacerse en C++.

La siguiente tabla muestra los modificadores permitidos, fuera de los modificadores de acceso, en Java y C# para los métodos.

Tabla 4.4. Modificadores de Acceso a Métodos en Java y C#

C#	Java	Descripción
Static	static	Método estático.
Abstract	abstract	Método abstracto, esto es, no implementado.
<sin equivalente>	final	Método que no puede ser sobrescrito en una clase derivada.
Extern	native	Método implementado externamente en otro lenguaje.
<sin equivalente>	synchronized	Método sincronizado.

Virtual	<sin equivalente>	Método que puede ser sobrescrito en una clase derivada.
Override	<sin equivalente>	Método que sobrescribe otro, declarado como virtual o abstract, en una clase base.
New	<sin equivalente>	Método que oculta otro en una clase base.
sealed override	<sin equivalente>	Método que sobrescribe otro, declarado como virtual o abstract, en una clase base, evitando también que vuelva a ser sobrescrito en una clase derivada.

A diferencia de C++ y Java, los métodos estáticos en C#, al igual que los datos miembros estáticos, no pueden ser llamados utilizando "this" desde dentro de la clase a la que pertenecen, y sólo pueden ser llamados utilizando el nombre de la clase desde fuera de ésta. En los tres lenguajes, desde un método estático no puede accederse a ningún elemento no-estático de la clase. Un buen ejemplo es el método Main de una clase ejecutable. El siguiente código en C# muestra este caso.

```
using System;
class MetodosEstaticos {
    public static void saludar() {
        Console.WriteLine("Hola");
    }
    public void despedir() {
        Console.WriteLine("Adios");
    }
}
class Principal {
    public static void Main(string[] args) {
        MetodosEstaticos.saludar();
        MetodosEstaticos.despedir();// ERROR
        MetodosEstaticos obj = new MetodosEstaticos();// CORRECCIÓN
        obj.despedir();
    }
}
```

En el código anterior, desde Main sólo puede llamarse directamente a "saludar", más aún, si la clase "Principal" tuviese otros miembros, sólo podrían accederse desde Main a los estáticos, dado que Main es estático. Para llamar al método "despedir" se requiere contar con una referencia a un objeto "MetodosEstaticos". Como contraparte, desde un método no-estático, o de instancia, sí se puede acceder a los miembros estáticos de la clase.

En Java y C#, un método abstracto no tiene implementación, dado que se espera que ésta sea dada en las clases que se hereden. Si un método es abstracto, la clase también deberá declararse como abstracta. Esto es similar a los métodos virtuales puros de C++.

En Java, un método con el modificador "final" imposibilita a las clases que heredan de sobrescribirlo. El equivalente más cercano en C# es "sealed override", con la diferencia que éste no se puede utilizar en la primera implementación del método, sino en una de sus sobrescrituras.

El modificador synchronized es utilizado en la programación concurrente y se verá en capítulos posteriores. El uso del modificador native va más allá de los alcances del curso.

Los modificadores relacionados con el comportamiento polimórfico de un método (new, virtual, abstract, override y sealed) se verán más adelante.

A diferencia de C++, no se pueden asignar valores por defecto a los parámetros.

El Paso de Parámetros

Los lenguajes de programación suelen diferenciar entre dos tipos de pasos de parámetros:

Paso por valor: El parámetro recibe una copia del dato original. Toda modificación al dato del parámetro no modificará al dato original.

Paso por referencia: El parámetro recibe una referencia del dato original. Toda modificación al dato del parámetro modificará al dato original.

Sin embargo, es importante anotar que el paso por referencia no es otra cosa que el paso por valor de un puntero o una referencia (de la forma como se entiende el término referencia en Java y C#, como una variable cuyos datos referencian a un objeto). El concepto de paso por referencia sirve para simplificar el trabajo del programador y evitar que éste tenga que lidiar con la lógica del manejo directo de punteros y los riesgos que esto conlleva. Teniendo esto presente, el resto de esta sección explica cómo manejan el paso de parámetros Java y C#.

En Java, cuando se llama a un método y se pasan parámetros, se siguen reglas fijas en el modo cómo estos parámetros son pasados. Todas las variables de tipo primitivo son pasadas por valor, mientras que las variables de tipo objeto (llamadas simplemente "referencias" en la jerga de Java) se pasan por referencia. Como consecuencia de esto, no es posible modificar el valor de una variable primitiva llamando a un método al que se le pase dicha variable como parámetro. Tampoco es posible hacer que una referencia apunte a otro objeto llamando a un método y pasando dicha referencia como parámetro. En el caso de los arreglos, éstos son objetos especiales en Java, por lo que su paso es por referencia.

En C# se ofrece mayor control en el paso de parámetros que en Java. Por defecto el paso de parámetros es como el indicado en Java. Adicionalmente se puede pasar "por referencia" las variables tipo valor (lo que incluye las variables primitivas) y "por referencia a referencia" las variables tipo referencia. Pasar "por referencia una referencia" es similar a pasar "un puntero a un puntero" en C++, sin sus complicaciones sintácticas. Para esta característica adicional de C# se utilizan las palabras reservadas "ref" y "out". El siguiente programa muestra el uso de estos tipos de pasos de argumentos en C#.

```
using  System;
class Principal {
    public static void mostrar(string mensaje, int[] arreglo, int indice) {
        Console.WriteLine(mensaje);
        Console.WriteLine("  arreglo.Length=" + arreglo.Length);
        Console.WriteLine("  arreglo[0]=" + arreglo[0]);
        Console.WriteLine("  arreglo[1]=" + arreglo[1]);
        Console.WriteLine("  arreglo[2]=" + arreglo[2]);
        Console.WriteLine("  indice=" + indice);
    }
    public static void modificar1(int[] arr, int indice) {
        arr[indice++] += 100;
        arr = new int[10];
    }
    public static void modificar2(ref int[] arr, ref int indice) {
        arr[indice++] += 100;
        arr = new int[10];
    }
    public static void obtener(out string nombre, out string apellido) {
        nombre = "Jose";
        apellido = "Perez";
    }
    public static void Main(string[] args) {
        int[] arreglo = {1,2,3};
        int indice = 0;
        mostrar("al inicio", arreglo, indice);
        modificar1(arreglo, indice);
        mostrar("luego de llamar a 'modificar1'", arreglo, indice);
        modificar2(ref arreglo, ref indice);
        mostrar("luego de llamar a 'modificar2'", arreglo, indice);
        string nombre, apellido;
        obtener(out nombre, out apellido);
        Console.WriteLine("nombre=" + nombre + ", apellido=" + apellido);
    }
}
```

A diferencia de C++, no es posible ni en Java ni en C# definir valores por defecto para los parámetros.

C# permite declarar una lista indeterminada de parámetros utilizando la palabra reservada "params". El parámetro declarado con "params" debe ser un arreglo y debe ser el último de la lista de parámetros del método. El siguiente código muestra el uso de params.

```
using System;
class Principal {
    public static void F(params int[] args) {
        Console.Write("args contiene {0} elementos:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }
    public static void Main(string[] args) {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

Como puede verse en el ejemplo anterior, la llamada a un método con un argumento "params" es más flexible que en C++ con los tres puntos "...". Se puede pasar tanto un arreglo como parámetro, como una secuencia de parámetros independientes. En este último caso, el lenguaje agrupa dichos argumentos en un arreglo pasándoselos como tal al método.

Las Propiedades y los Indizadores

Es común definir métodos de acceso y obtención (set/get) a los datos encapsulados u ocultos por una clase. Estos métodos permiten contar con bloques de código que pueden controlar que dicho acceso se realice de manera conveniente, por ejemplo, que no se pueda establecer un valor incorrecto a un dato. Como contraparte, estos métodos pueden quitarle una significativa legibilidad al código que los utiliza.

Por ejemplo, supongamos que tenemos una clase que implementa un número complejo, y quisiéramos multiplicar su parte real por dos. Utilizando métodos getReal/setReal tendríamos un código como el siguiente:

```
Complejo c = new Complejo(1,2);
c.set( c.get() * 2);
```

Lo que es significativamente menos legible si tuviésemos acceso directo al dato miembro real y codificáramos:

```
Complejo c = new Complejo(1,2);
c.real = c.real * 2;
```

La ilegibilidad aumenta conforme la expresión se haga más compleja. Más aún, si las clases que utilizamos no guardan un estándar, el programador que hace uso de ellas requerirá encontrar el nombre de dos funciones para manejar el mismo dato.

Como una manera de simplificar el acceso a los datos de una clase, C# define el concepto de "propiedad". Una propiedad representa a uno o dos métodos de acceso a un dato miembro (o uno calculado) de una clase. La sintaxis de una propiedad es:

```
[modificadores] <tipo de dato> <nombre de la propiedad> {
    get { <cuerpo del bloque get> }
    set { <cuerpo del bloque set> }
}
```

Se puede definir sólo el bloque set (con lo que tendríamos una propiedad de sólo escritura), sólo el bloque get (propiedad de sólo lectura) o ambos bloques (propiedad de lectura y escritura). Como puede verse en la sintaxis, las propiedades pueden recibir modificadores al igual que los datos miembros y los métodos.

Por ejemplo, la siguiente clase en C# implementa el acceso a un archivo sólo para lectura. Debido a esto, el dato del tamaño del archivo nunca se cambiará, por lo que este dato puede implementarse como una propiedad de sólo lectura. Por otra parte, la posición actual de lectura del archivo sí puede desearse que sea leída o modificada por el programa que usa esta clase, por lo que puede implementarse como una propiedad de lectura y escritura.

```
class ArchivoSoloLectura {
    ...
    public int Tamano {
        get {
            // aquí va el código que permite obtener el tamaño del archivo
            return TamanoCalculado;
        }
    }
    public int Posicion {
        set {
            // asumiendo que el método interno (privado) establecerPosicion
            // realiza el trabajo de llamar a las librerías del sistema
            // operativo para recolocar el puntero a un archivo
            establecerPosicion( value );
        }
        get {
            // aqui íría el código que permitiría calcular la posición actual
            return PosicionCalculada;
        }
    }
}
```

Note que la palabra reservada "value" es utilizada dentro del bloque "set" para hacer referencia al valor que se le quiere establecer a la propiedad al utilizarla en un programa. El siguiente código hace uso de esta clase.

```
ArchivoSoloLectura arch = new ...;
Console.Write("El tamano del archivo es " + Arch.Tamano);
// Aquí se realizaría algún trabajo de lectura
Console.Write("La posición actual en el archivo es " + Arch.Posicion);
Arch.Posicion = 0; // se retorna al inicio del archivo
```

La última línea del programa anterior provoca una llamada al bloque set de la propiedad Posicion, dentro del cual la palabra reservada "value" contendría en valor "0".

Bajo el mismo concepto de la claridad del código brindado por las propiedades, cuando un objeto representa una colección de elementos (por ejemplo, como una lista enlazada) sería más claro su manejo si fuera posible acceder a estos elementos utilizando la misma sintaxis que con los arreglos. Para esto, C# define los **indizadores**.

Un indizador es como una propiedad pero para manejar un objeto con la misma sintaxis que la de un arreglo unidimensional. La sintaxis de declaración de un indizador es como sigue.

```
[modificadores] <tipo de dato> this[<tipo del indice> <nombre del indice>] {
    get { <cuerpo del bloque get> }
    set { <cuerpo del bloque set> }
}
```

Los indizadores no tienen un nombre particular, como las propiedades, puesto que son llamados automáticamente cuando se utiliza la sintaxis del acceso a los elementos de un arreglo con ellos. El parámetro entre corchetes sirve para declarar el tipo y nombre de la variable que servirá para hacer referencia al índice (o lo que equivalga al mismo) dentro de los bloques set y get. Esto implica que no requerimos necesariamente utilizar un dato de tipo entero como índice, sino cualquier tipo de dato que deseemos.

Como ejemplo, imaginemos que deseamos crear una clase que maneje una lista enlazada de objetos Nodo. Adicionalmente, queremos que el programador tenga la posibilidad de recorrer los elementos de

esta lista de la misma forma como lo haría con un arreglo, por simplicidad. Tendríamos el siguiente esqueleto del código de esta clase.

```
class ListaEnlazada {
    // Acá van los datos miembros de la clase y sus métodos correspondientes al
    // manejo de una lista enlazada. Se asume que la clase "Nodo" ya esta
    // implementada
    Nodo primerNodo;

    // Declaración del indizador
    public Nodo this [ int indice ] {
        set {
            Nodo unNodo = primerNodo;
            for( int i = 0; i < indice; i++)
                   unNodo = unNodo.siguiente();
            unNodo.valor = value;
        }
        get {
            Nodo unNodo = primerNodo;
            for( int i = 0; i < indice; i++)
                   unNodo = unNodo.siguiente();
            return unNodo.valor;
        }
    }

    // Declaración de una propiedad
    public int Tamano {
        get {
            // Aquí se recorre la lista contándose los nodos hasta llegar al final
            return TamanoCalculado;
        }
    }
}
```

Note en el ejemplo anterior el uso del parámetro utilizado como índice dentro de los bloques get y set del indizador. Note además que es posible definir propiedades en una clase con un indizador. El siguiente programa hace uso de esta clase.

```
ListaEnlazada lista = ...;
// Aquí se inicializa la lista con valores para sus nodos
for( int i = 0; i < lista.Tamano; i++ )
    // Acá la expresión "lista[i]" equivale a una llamada al bloque "get"
    // del indizador de la clase ListaEnlazada
    Console.WriteLine("Valor en la posición " + i + " = " + lista[i]);
```

Note como en el ejemplo anterior se accede a la variable lista como si se tratase de un arreglo. Otro ejemplo del uso de indizadores y propiedades es la clase "string" de C#, la que implementa el acceso a los carácteres de la cadena como un indizador de sólo lectura, así como la propiedad de sólo lectura Length para obtener su longitud.

Las Estructuras

C++ permite definir tipos de datos "estructura" y "clase". Las estructuras son herencia de C, donde se utilizaban como forma de crear nuevos tipos de datos compuestos. Sin embargo, la implementación interna de C++ para las estructuras es casi la misma que para las clases, con la única diferencia que por defecto los miembros de una estructura son públicos, por lo que la decisión de mantener la palabra reservada "struct" en C++ responde solo a razones de compatibilidad.

C# también permite definir estructuras como un tipo especial de clases. Las estructuras en C# están diseñadas para ser tipos de datos compuestos más limitados que el resto de clases pero más eficientes que éstas. A continuación se listan las capacidades de las estructuras en C#:

- Sus variables son reservadas automáticamente en memoria de pila y se manejan como tipos de dato valor.

- No permiten la herencia de otras estructuras ni clases, pero sí la implementación de interfaces (lo que se verá más adelante).

- Permiten la definición de métodos, pero no el polimorfismo.

- Permiten la definición de constructores, menos los constructores por defecto, dado que el compilador siempre incluye uno.

- No permiten la inicialización de sus datos miembros de instancia en la misma declaración.

- No se puede utilizar una variable estructura hasta que todos sus campos hayan sido inicializados.

Una variable tipo estructura en C# se reserva automáticamente en pila, mientras que en C++ el programador es quien decide si se almacena en pila o en montón.

El uso de estructuras en C# es recomendado cuando se cumple una o más de las siguientes condiciones:

- Cuando la información que contendrán las estructuras será manipulada como un tipo de dato primitivo.

- Cuando la información que se almacena es pequeña, esto es, menor o igual a 16 bytes.

- Cuando no se requiere utilizar herencia y/o polimorfismo.

La sintaxis de declaración de una estructura en C# es:.

```
[modificadores] struct <nombre> [: <lista de interfaces que implementan>]
{   <datos,tipos,funciones>
}
```

El siguiente código corresponde a un ejemplo de definición y uso de una estructura.

```
using System;
struct Punto {
    public int x, y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public string Descripcion() {
        return "(" + x + "," + y + ")";
    }
}
class PruebaDeEstructuras
{
    public static void Main(string[] args)
    {
        Punto p1 = new Punto();
        Punto p2 = new Punto(10, 20);
        Console.WriteLine("p1 = " + p1.Descripcion());
        Console.WriteLine("p2 = " + p2.Descripcion());
        Punto p3;
        Console.WriteLine("p3 = " + p3.Descripcion()); // ERROR
        p3.x = 30;
        p3.y = 40;
        Console.WriteLine("p3 = " + p3.Descripcion());
    }
}
```

Puede observarse en la inicialización de la variable "p1" que, efectivamente, el compilador agrega un constructor por defecto "siempre" a la estructura. Este constructor por defecto inicializa los datos miembros del objeto estructura a sus valores por defecto. La variable "p3" se intenta utilizar sin haber sido inicializada, lo que arroja un error. Enseguida se procede a inicializar individualmente cada dato de la estructura. Sólo cuando todos los datos de la estructura "p3" fueron inicializados es posible utilizar la variable "p3" para otros fines, como por ejemplo llamar a su método "Descripcion". Note, del uso de la variable "p3", que no se requiere utilizar el operador new para inicializar una variable estructura, más aún, luego de ejecutar la línea que declara la variable "p3" la memoria de este objeto estructura ya está reservada en pila. Por tanto, el operador new sólo permite llamar a un constructor para inicializar la variable, y no reservarle memoria como sí ocurre con las clases declaradas como "class".

La Herencia y el Polimorfismo

Cuando un tipo de dato extiende la definición de otro, se dice que el primero hereda del segundo. En esta sección revisaremos el manejo de la herencia en los tres lenguajes estudiados, así como las capacidades que cada uno ofrece en cuanto al manejo del polimorfismo.

La Herencia

Un tipo de dato hereda de otro todos sus miembros: Datos, métodos y otros tipos de datos. El que un tipo de dato herede de otro es una característica que facilita el desarrollo de programas gradualmente más complejos.

A la clase de la que se hereda se le llama clase base o súperclase. A la clase que hereda de otra se le llama clase derivada o subclase. Cuando una clase hereda de más de una base, se le conoce como herencia múltiple. Como veremos más adelante, la herencia múltiple tiene una serie de beneficios y problemas, por lo que no es soportada en algunos lenguajes orientados a objetos.

Esta herencia puede producir diferentes tipos de conflicto:

- Conflicto de espacio de nombres: Dado que los miembros de la clase base pasan a formar parte del espacio de nombres de la clase derivada, existe el problema potencial de que un miembro de una clase base coincida en nombre con uno de la clase derivada.

- Conflicto en la resolución de las llamadas a los métodos: Dado que es posible diferenciar a un método de otro por otros elementos además de su nombre, es posible definir más de un método con el mismo nombre, tanto en la clase base como en la derivada, por lo que se requiere de una estrategia para determinar a qué método se está llamando realmente dentro del contexto de la llamada. Esta situación se conoce como "polimorfismo".

- Problemas de duplicidad de datos: Lo que es un problema cuando se tiene herencia múltiple de clases que a su vez heredan, directa o indirectamente, de una misma clase base. En dichos casos, los objetos de la clase más derivada contendrán datos miembros duplicados, uno por cada rama de la herencia.

La tabla 4.5 muestra con qué tipos de datos es posible utilizar herencia y qué tipo de herencia es soportada:

Tabla 4.5. Tipos de herencia por lenguaje

	C++	Java	C#
Tipos de datos	Clases y estructuras	Clases e interfaces	Clases e interfaces
Tipo de herencia	Herencia simple y múltiple	Herencia simple de clases y múltiple	Herencia simple de clases y

		de interfaces	múltiple de interfaces

La Declaración de la Herencia

La sintaxis de declaración de la herencia en C++ es:

```
class <nombre>: [mod.  de acceso]<clase base1>, [mod.  de acceso]<clase base2>, ...
{ <cuerpo de la clase> };
```

La sintaxis de declaración de la herencia en Java es:

```
class <nombre> extends <clase base> implements <interfaz1>, <interfaz2>, ...
{ <cuerpo de la clase> }
```

La sintaxis de declaración de la herencia en C# es:

```
class <nombre>: <clase base>, <interfaz1>, <interfaz2>, ...
{ <cuerpo de la clase> }
```

Los modificadores de acceso permitidos en la declaración de la herencia en C++ son private, protected y public.

El Proceso de Construcción y Destrucción

Cuando una clase hereda de otra, el proceso de construcción y destrucción de un objeto sigue la siguiente lógica:

- Durante la construcción, se llama uno a uno a los constructores, desde las dases base y avanzando por el árbol de herencia hacia las clases más derivadas. Esto permite que el código de construcción de un objeto derivado se ejecute dentro de un contexto en el que se asegure que sus datos heredados están inicializados y en un estado consistente.

- Como contraparte, durante la destrucción, se llama uno a uno a los destructores, desde las clases más derivadas y avanzando por el árbol de herencia hacia las clases base. Esto permite que el código de destrucción de un objeto derivado se ejecute dentro de un contexto en el que se asegure que sus datos heredados aún existen y contienen valores consistentes.

Cuando las clases, en un árbol de herencia, contienen constructores por defecto, la secuencia de llamada durante el proceso de construcción de un objeto se realiza automáticamente. Sin embargo, cuando una clase base no posee un constructor por defecto o se desea ejecutar determinado constructor con parámetros, se puede especificar dicho paso de parámetros desde el constructor de la clase derivada, de la siguiente forma:

C++:

```
class Base {...};
class Derivada: public Base{
    public Derivada(...) : Base(...)
    {...}
};
```

Java:

```
class Base {...}
class Derivada extends Base{
    public Derivada(...)
    { super(...);  ...  }
}
```

C#:

```
class Base{...}
class Derivada: Base{
```

```
    public Derivada(...): base(...)
    {...}
}
```

Note que mientras en C++ se especifica este paso de parámetros, desde el constructor de una clase derivada al de una clase base, con el nombre mismo de la clase base, en Java y C# se utilizan las palabras reservadas "super" y "base" respectivamente. Esto se debe a que C++ soporta herencia múltiple, por lo que podría requerirse pasar parámetros a más de un constructor, mientras que Java y C# soportan sólo herencia simple de clases. La herencia múltiple se verá más adelante.

Note además que la llamada a "super" en Java debe hacerse obligatoriamente en la primera línea del constructor.

Acceso a los Miembros Heredados

En la herencia, la clase derivada puede contener miembros que coincidan en nombre con algunos de la base. Esto no es un error, y es aceptado en los tres lenguajes estudiados. Sin embargo, es necesario contar con un mecanismo para diferenciar, en estos casos, a qué miembro se refiere un pedazo de código, tanto dentro como fuera de la clase.

Dentro de los métodos de la clase derivada, cuando se utiliza directamente el nombre del miembro en conflicto, el compilador asume que nos referimos al de la clase derivada. Para referirnos al de la clase base debemos utilizar una sintaxis especial:

- En C++: <nombre de la clase base> **::** <nombre del miembro>
- En Java: **super.**<nombre del miembro>
- En C#: **base.**<nombre del miembro>

Como puede verse, en Java y C# sólo puede accederse al miembro heredado de la clase base. El siguiente ejemplo en Java muestra esta limitación:

```
class Base {
    void m() { System.out.println("Llamada a Base.m"); }
}
class Derivada1 extends Base {
    void m() { System.out.println("Llamada a Derivada1.m"); }
}
class Derivada2 extends Derivada1 {
    void m() { System.out.println("Llamada a Derivada2.m"); }
    void prueba() {
        m();
        super.m();
        super.super.m(); // ERROR: Llamada inválida
    }
}
public class PruebaDeHerencia1 {
     public static void main(String[] args) {
        Derivada2 obj = new Derivada2();
        obj.prueba();
        obj.m();
     }
}
```

Como se ve en el ejemplo, no es posible acceder a la implementación del método "m" en "Base" desde un método de la clase "Derivada2", sólo a la implementación en "Derivada1", su clase base inmediata. Asimismo, la llamada a "m" desde "main" ejecutará la última implementación de este método, la de la clase "Derivada2". Como veremos más adelante, esto es un tipo de polimorfismo.

El Polimorfismo

El polimorfismo ocurre cuando la llamada a un método se resuelve sobre la base del contexto de la llamada, dado que existe más de una implementación para dicho método, es decir, la llamada a un método puede tomar distintas formas, según como ésta se realice. Veremos tres casos de polimorfismo:

- La sobrecarga de funciones y métodos.
- La sobrescritura en la herencia.
- La sobrescritura en la implementación de las interfaces.

La Sobrecarga

La forma más simple de polimorfismo es la sobrecarga, en donde dos métodos con el mismo nombre son diferenciados por el lenguaje basándose únicamente en sus argumentos. En este sentido, es el compilador quien resuelve a qué método se llamará, cuando compila y genera el código objeto del programa, sobre la base de los argumentos que se le pasan al llamar un método o función global (en el caso de C++).

El siguiente ejemplo muestra la sobrecarga en C#.

```
using System;

struct Punto {
    public int x, y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Figura {
    Punto posicion;
    public Figura() : this(new Punto()) {
    }
    public Figura(Punto p) {
        Mover(p);
    }
    public void Mover(int x, int y) {
        Mover(new Punto(x, y));
    }
    public void Mover(Punto p) {
        posicion = p;
    }
}
class Circulo : Figura {
    double radio;
    public Circulo(double r) : this(new Punto(), r) {
    }
    public Circulo(Punto p, double r) : base(p) {
        radio = r;
    }
    public void Mover(Punto p, double r) {
        Mover(p);
        radio = r;
    }
}
```

En el ejemplo anterior el método “Mover” es sobrecargado tres veces, dos en la clase “Figura” y una en la clase “Circulo”. Además, los constructores de ambas clases están sobrecargados. Cuando se llama al método “Mover” se resuelve dicha llamada sobre la base del tipo de la variable utilizada y los parámetros pasados. El siguiente código utiliza las clases anteriores.

```
Figura f = new Figura();
f.Mover(2,3);
```

```
Circulo c = new Circulo(10);
c.Mover(new Punto(7,4), 2);
f.Mover(new Punto(7,4), 2); // ERROR
```

La última línea anterior genera un error debido a que "f" es tipo "Figura" y dentro de esta clase no existe una sobrecarga apropiada para los argumentos pasados en la llamada.

Es importante notar que dos métodos no pueden diferenciarse sólo por su valor de retorno, por lo que dos métodos de la misma sobrecarga con los mismos argumentos deberán tener necesariamente el mismo valor de retorno. La sobrecarga es soportada por C++ y C#. Como se verá en breve, Java sólo permite sobrecargas de un tipo especial.

Adicionalmente C++ y C# permiten la sobrecarga de operadores. C++ ofrece una amplia gama de opciones en cuanto al juego de operadores que pueden ser sobrecargados y la forma en que puede declararse dichas sobrecargas. C# por el contrario, ofrece un conjunto restringido de operadores que pueden ser sobrecargados y un único formato de declaración de dicha sobrecarga. Por ejemplo, la sobrecarga del operador de suma en C# tiene el siguiente formato:

```
public static Tipo operator+(Tipo1 op1, Tipo2 op2) { ...  }
```

El siguiente código utiliza una sobrecarga del operador "+" para una clase Vector:

```
class Vector {
    private double x, y;
    public Vector(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public static Vector operator+(Vector op1, Vector op2) {
        return new Vector(op1.x + op2.x, op1.y + op2.y);
    }
    public void Imprimir() {
        Console.WriteLine("Vector[x={0}, y={1}]", x, y);
    }
}
class PruebaDeSobrecargaDeOperadores {
    public static void Main(string[] args) {
        Vector a = new Vector(1, 2), b = new Vector(3, 4), c;
        c = a + b;
        c.Imprimir();
    }
}
```

La declaración de una sobrecarga debe ser pública y estática y el tipo del primer parámetro debe coincidir con el tipo de la clase dentro de la que se declara la sobrecarga.

El Ocultamiento

La sobrecarga funciona bien cuando podemos diferenciar dos métodos por sus argumentos, pero es posible tener un método en una clase base con el mismo nombre y los mismos argumentos en la clase derivada. Esto es deseable cuando lo que deseamos conseguir es un **ocultamiento** de la implementación original de dicho método, de forma que en el código que utilice nuestra clase se llame a esta nueva implementación. El ocultamiento es un tipo de **sobrescritura**, y es tan eficiente como la sobrecarga.

El siguiente programa en C++ muestra el uso del ocultamiento.

```
class Base {
    public: void Metodo( ) { cout << "Base::Metodo\n"; }
};
class Derivada : public Base {
    public: void Metodo( ) { cout << "Deri::Metodo\n"; }
};
void main() {
    Base ObjBase; Derivada ObjDerivada; Base* pBase;
    ObjBase.Metodo( );
```

```
    ObjDerivada.Metodo( );
    pBase = &ObjBase;
    pBase->Metodo( );
    pBase = &ObjDerivada;
    pBase->Metodo( );
}
```

Al ejecutar este programa se ve que las dos primeras llamadas a "Metodo" utilizando las variables "ObjBase" y "ObjDerivada" ejecutan la implementación respectiva en cada clase. En estos casos, el compilador ya no ha podido basarse en los argumentos de la llamada al método para decidir a cuál implementación llamar, sino en el tipo de las variables utilizadas. Las dos últimas llamadas utilizan un puntero del tipo de la clase base "Base" para, apuntando a cada objeto, llamar al "Metodo". Ambas llamadas se resuelven hacia la implementación en la clase "Base", dado que, al igual que en las dos primeras llamadas, el compilador se basó en el tipo de la variable utilizada para decidir a qué implementación llamar.

El ejemplo anterior muestra una clara ventaja y limitación del ocultamiento. Si bien el ocultamiento es eficiente dado que la resolución de la llamada se realiza en tiempo de compilación, la capacidad de ocultar es limitada sólo a los casos donde el tipo del objeto creado coincide con el tipo de la variable que lo referencia o apunta. Este es el caso del puntero de tipo "Base" utilizado para llamar al método "Metodo" de un objeto de tipo "Derivada". Aquí la nueva implementación de "Metodo" no ocultó a la implementación original, lo que probablemente se desea que ocurra.

El siguiente programa corresponde a una versión en C# del programa anterior.

```
using System;
class Base {
    public void Metodo( ) {
        Console.WriteLine("Base.Metodo");
    }
}
class Derivada : Base {
    public new void Metodo( ) {
        Console.WriteLine("Derivada.Metodo");
    }
}
class PruebaDeEstructuras {
    public static void Main(string[] args) {
        Base ObjBase = new Base();
        Derivada ObjDerivada = new Derivada();
        ObjBase.Metodo( );
        ObjDerivada.Metodo( );
        Base refBase;
        refBase = ObjBase;
        refBase.Metodo( );
        refBase = ObjDerivada;
        refBase.Metodo( );
    }
}
```

Otra consecuencia de la forma cómo funciona el ocultamiento es que no se permite que las implementaciones en la clase base utilicen las versiones actualizadas de los métodos ocultados. Los métodos en la base siempre trabajan con "lo conocido" para su clase. El siguiente ejemplo en C++ muestra este caso.

```
class Base {
  public: void Met1() { cout << "Base::Met1\n"; }
          void Met2() { Met1(); }
};
class Derivada : public Base {
  public: void Met1() { cout << "Deri::Met1\n"; }
};
void main() {
    Base ObjB; ObjB.Met2();
    Derivada ObjD; ObjD.Met2();
}
```

En el ejemplo mostrado, el ocultamiento de "Met1" no es aprovechado por "Met2", por lo que el texto mostrado será "Base::Met1", cuando lo que quizá se deseaba era "Deri::Met1".

Como puede observarse, la implementación del método "Metodo" en la clase "Derivada" requiere el uso de la palabra reservada **new**. C# busca con esto que sea claro que dicho método está ocultando a otro en alguna de las bases del árbol de herencia.

En resumen, el ocultamiento es eficiente, dado que se resuelve en tiempo de compilación, pero puede originar llamadas incorrectas a métodos sobrescritos cuando los objetos son tratados con variables cuyo tipo corresponda a alguna de las clases base del objeto. Para solucionar esto, el programa debería analizar en tiempo de ejecución, y ya no durante la compilación, a qué objeto verdaderamente apunta o referencia una variable, y sobre la base de esto decidir a qué método llamar. Esto es lo que realiza la **sobrescritura virtual**.

La Sobrescritura Virtual

Cuando un método se sobrescribe virtualmente el compilador agrega información adicional a la clase, la que es utilizada en tiempo de ejecución para determinar a que tipo de objeto se está apuntando realmente y por tanto, a qué método se debe de llamar.

Las siguientes sintaxis corresponden a la manera cómo declarar una sobrescritura virtual, en cada lenguaje:

- C++: Se coloca "virtual" en la base.

```
class Base { ...  virtual void Met( ) {...} ...  };
```

- Java: No existe sobrecarga ni ocultamiento. El polimorfismo siempre es mediante sobrescritura virtual.

- C#: Se coloca "virtual" en la base y "override" en la derivada.

```
class Base { ...  virtual void Met( ) {...} ...  };
class Deri : Base { ...  override void Met( ) {...} ...  };
```

En resumen, la sobrescritura virtual permite que desde cualquier método de la clase base, de la clase derivada o desde fuera de ellas, una llamada a un método se resuelva sobre la base del tipo del objeto apuntado o referenciado, no el de la variable que lo apunta o referencia. Como es de suponerse, la sobrescritura virtual es menos eficiente que el ocultamiento.

La Implementación de Interfaces

En general se suele decir que la interfaz de una clase se refiere al conjunto de métodos que ésta expone públicamente, para el manejo de los objetos de la misma.

El concepto de interfaz no está relacionado a la implementación de los métodos de una clase, sólo a la declaración de los mismos. De esta forma, se puede encontrar que varias clases, no relacionadas en una herencia, pueden exponer interfaces similares. Por tanto, si se pudiera tipificar y homogenizar los elementos similares de las interfaces de varias clases, podrían manejarse sus objetos de manera homogénea, aún cuando no pertenezcan a una misma herencia.

Esta tipificación de la interfaz de una clase se realiza en Java y C# mediante la palabra reservada "interface". Las sintaxis correspondientes a la declaración de una interfaz en Java y C# son:

En Java:

```
[modificadores] interface <nombre> [ extends <lista de interfaces> ]
{ <declaración de los métodos> }
```

En C#:

```
[modificadores] interface <nombre> [ : <lista de interfaces> ]
{ <declaración de los métodos> }
```

Las interfaces soportan la herencia múltiple, debido a que no puede existir conflicto de implementación (dado que nada se implementa) ni tampoco duplicidad de datos (dado que no se permiten declarar datos). En C++ no existe el concepto de interfaz, pero puede ser simulado mediante una clase que contenga sólo métodos virtuales puros públicos.

Una interfaz declara, no implementa, un conjunto de métodos que corresponden a una funcionalidad que una clase puede exponer. Por ejemplo, la interfaz "dibujable" puede ser implementada tanto por una clase "marquesina" como una clase "fotografía". Por tanto, las interfaces agrupan a un conjunto de métodos públicos que pueden ser implementados por una clase.

El siguiente programa Java hace uso de una interfaz.

```
interface Dibujable {
  void Dibujar( );
}
class Imagen implements Dibujable {
  public void Dibujar( ) { System.out.println("Llamando a Dibujar en Imagen"); }
}
class Texto implements Dibujable {
  public void Dibujar( ) { System.out.println("Llamando a Dibujar en Texto"); }
}
public class PruebaDeInterfaces {
    public static void main(String[] args) {
       Imagen img = new Imagen();
       Texto txt = new Texto();
       Dibujable dib = img;
       dib.Dibujar();
       dib = txt;
       dib.Dibujar();
    }
}
```

El mismo programa en C# sería:

```
using System;
interface Dibujable {
   void Dibujar( );
}
class Imagen : Dibujable {
   public void Dibujar( ) {   Console.WriteLine("Llamando a Dibujar en Imagen");
}
}
class Texto : Dibujable {
   public void Dibujar( ) { Console.WriteLine("Llamando a Dibujar en Texto"); }
}
public class PruebaDeInterfaces {
   public static void Main() {
       Imagen img = new Imagen();
       Texto txt = new Texto();
       Dibujable dib = img;
       dib.Dibujar();
       dib = txt;
       dib.Dibujar();
    }
}
```

El mismo programa en C++ sería:

```
class Dibujable {
   public: virtual void Dibujar( ) = 0;
};
class Imagen: public Dibujable {
```

```
    public: virtual void Dibujar( ) { cout << "Llamando a Dibujar en Imagen"; }
};
class Texto: public Dibujable {
    public: virtual void Dibujar( ) { cout << "Llamando a Dibujar en Texto"; }
};
void main() {
    Imagen* img = new Imagen();
    Texto* txt = new Texto();
    Dibujable* dib = img;
    dib->Dibujar();
    dib = txt;
    dib->Dibujar();
}
```

Una clase puede implementar más de una interfaz y en el caso de C#, una estructura también. Como puede verse, el concepto de herencia múltiple de interfaces en Java y C# reemplaza al de herencia múltiple de clases en C++, evitando los problemas que ésta última tiene.

El siguiente código de programa en C# muestra una herencia múltiple de interfaces.

```
interface IArchivo {
    void posicion( );
}
interface IArchivoBinario : IArchivo {
    byte leerByte( );
    void escribirByte(byte b);
}
interface IArchivoTexto : IArchivo {
    char leerChar( );
    void escribirChar(char b);
}
class ArchivoBinario : IArchivoBinario { ...  }
class ArchivoTexto : IArchivoTexto { ...  }
class ArchivoBinarioTexto : IArchivoBinario, IArchivoTexto { ...  }
```

Aún cuando no se declare como públicos los métodos de una interfaz, éstos siempre lo son. Por tanto, las implementaciones de estos métodos en las clases deberán ser declaradas como públicas.

Las Clases Abstractas

Cuando una clase no implementa un método se dice que ésta es abstracta. Dado que las clases abstractas son clases "a medio implementar", éstas no pueden utilizarse para instanciar objetos, pero sí como clases base de otras clases que sí implementen dichos métodos faltantes.

Declaración de una Clase Abstracta

En C++ las clases que contienen métodos virtuales puros son implícitamente abstractas. En Java y C#, las clases que sólo declaren algunos métodos, sin implementarlos, deben ser declaradas explícitamente mediante la palabra reservada "abstract" como un modificador más en la declaración de la clase.

Es importante recalcar que una clase abstracta se diferencia de una interfaz en que puede contener métodos no abstractos y por tanto, funcionalidad implementada.

El siguiente programa en C# muestra el uso de una clase abstracta.

```
using System;
abstract class Base{
    public void imprimir() {Console.WriteLine("Imprimir: Clase Base");}
    abstract public void metodo();
}
class Derivada : Base{
    new public void imprimir(){Console.WriteLine("Imprimir: Clase Derivada");}
    override public void metodo(){Console.WriteLine("Metodo: Clase Derivada");}
}
class MainClass{
```

```
    public static void Main(string[] args){
        Base objBase ;//= new Base(); // si se descomenta, habria un error
                                        // pues no se puede instanciar la clase
                                        // por ser abstracta
        Derivada objDerivada = new Derivada();
        objbase = objDerivada;
        objBase.imprimir(); objBase.metodo();
        objDerivada.imprimir(); objDerivada.metodo();
    }
}
```

Note que en C# el método abstracto en la clase base debe de declararse explícitamente como “abstract”, lo que no requiere Java. Note también que la implementación del método en la clase derivada se declara como “override”, debido a que los métodos abstractos son implícitamente virtuales. Como en Java todo polimorfismo es virtual, un equivalente de este programa en Java no requeriría utilizar ninguna palabra reservada especial en la declaración de la implementación en la clase derivada.

Todo método declarado dentro de una interfaz en Java y C# es implícitamente abstracto, virtual y público.

Java permite la declaración de constantes dentro de sus interfaces. C# permite declarar propiedades e indizadores. La siguiente interfaz en C# declara una propiedad de sólo lectura y un indizador de lectura y escritura.

```
class UnaInterface {
    int UnaPropiedad { get; }
    int this [ int indice ] { get; set; }
}
```

Diferencias entre las Interfaces y las Clases Abstractas

Las clases abstractas son adecuadas cuando se desea tener una implementación básica común para clases derivadas que deberán implementar la funcionalidad faltante y posiblemente, sobrescribir parte de la básica. Por otro lado, las interfaces son adecuadas cuando no se requiere contar con una implementación básica común, dado que toda se realizará en las clases que las implementen. La tabla 4.6 sumariza estas diferencias.

Tabla 4.6. Diferencias entre las clases abstractas y las interfaces

	Interfaz	Clase abstracta
Declara métodos abstractos	Sí	Sí
Implementar métodos	No	Sí
Añadir datos miembros	No	Sí
Crear objetos	No	No
Crear arreglos y referencias	Sí	Sí

Los Tipos de Datos Anidados

Hasta el momento sólo se han estudiado los datos y las funciones como miembros de una clase, pero una clase también puede contener la definición de tipos como miembros. A un tipo de dato declarado dentro de otro tipo de dato se le conoce como tipo de dato anidado o “inner”, y al tipo que lo contiene “outer”.

En esta sección nos concentraremos en la declaración y uso de las clases anidadas o clases inner.

La Declaración de Clases Anidadas

Una clase anidada o inner se declara como cualquier otra clase, con la diferencia que, dado que son miembros de una clase, pueden aplicárseles modificadores propios de los miembros de éstas. Por

ejemplo, es posible declarar una clase inner como privada, por lo que sólo se podrán declarar variables de ésta dentro de la clase outer y ser manipulados desde los métodos de ésta.

Se puede tener varios niveles de anidación en la declaración de clases, es decir, una clase inner puede ser anidar otras clases inner. Una clase que no es inner de ninguna otra se le llama clase de alto nivel o "top-level".

Las clases anidadas son útiles cuando:

- Los objetos de dichas clases sólo van a ser utilizados dentro de su clase outer. Estas clases inner funcionarían como una librería interna de la clase outer. Sin embargo, si es posible que estas clases inner sean reutilizables fuera de la clase outer, es recomendable sacarlas de la clase outer y definirlas en un espacio de nombres independiente. La definición y uso de los espacios de nombres se verá más adelante.

- Los objetos de dichas clases requieren trabajar intensamente con los datos de la clase outer. Dado que la clase inner está definido dentro del ámbito de la clase outer, tiene acceso directo a todos los miembros de ésta, aún a los privados, lo cual puede ser beneficioso si es que el trabajo entre ambas clases es muy intenso.

- Cuando carezca de sentido crear objetos de esta clase sin que existan previamente objetos de su clase outer. En este caso, se requiere de un objeto de la clase outer para poder crear, sobre la base de su información interna y otros datos externos, objetos de la clase inner.

- Cuando la clase outer le da sentido a la inner. En este caso, la clase outer funciona como un espacio de nombres. Al igual que en el primer caso, si no se cumpliera esta condición se recomienda utilizar clases en un espacio de nombres independiente en lugar de clases inner.

El siguiente programa en Java muestra el uso de una clase inner.

```
class A {
    class B {
        public void metodo1() {
            System.out.println("Llamada a B.metodo1");
            metodo2();
        }
    }
    private void metodo2() {
        System.out.println("Llamada a A.metodo2");
    }
    public void metodo3() {
        System.out.println("Llamada a A.metodo3");
        B objB = new B();
        objB.metodo1();
    }
}
public class PruebaDeInterfaces {
     public static void main(String[] args) {
          A objA = new A();
          objA.metodo3();
     }
}
```

En el ejemplo anterior, la clase B es inner de la clase A, por lo que puede acceder a todos los miembros de ésta última, inclusive los privados, como la llamada al método "metodo2" desde su implementación de "metodo1". Dado que la clase B es un miembro más de la clase A, al no habérsele especificado un modificador de acceso posee el modificador de-paquete. Por tanto, es posible acceder desde "main" a dicha clase, declarar variables y crear objetos con ella.

La Creación de Objetos de Clases Anidadas

En el ejemplo anterior se creó un objeto de la clase B dentro del "metodo3" de la misma forma como se crean objetos de clases no inner. Sin embargo, para crear objetos inner fuera de su clase outer la sintaxis es diferente. El siguiente código puede haberse incluido en "main" para crear un objeto de la clase B:

```
A.B objB = objA.new B();
objB.metodo1();
```

Es interesante notar como el nombre de la clase outer forma parte del nombre de la clase inner, es decir, fuera de la clase A, el nombre completo de la clase B es "A.B". El uso de la palabra reservada "new" también requiere una sintaxis especial, dado que los objetos de la clase B son considerados "de instancia", esto es, se requiere utilizar una instancia de la clase A para crear una de la B. Piense en esto: Si la instancia del objeto referenciado por "objB" no hubiera sido creada haciendo referencia a un objeto de la clase A, ¿al método de qué objeto se estaría accediendo en la llamada que hace "metodo1" a "metodo2", dado que éste último es igualmente un método de instancia? Luego, cuando se crea un objeto de una clase inner de instancia, dicho objeto conserva una referencia al objeto outer en base al que fue creado, de forma que pueda acceder tanto a sus miembros estáticos como no-estáticos. Sin embargo, si esta característica no se deseara, la clase B pudo declararse como estática, es decir:

```
class A {
    static class B {
        public void metodo1() { ...  }
    }
    private static void metodo2() { ...  }
    public void metodo3() { ...  }
}
public class PruebaDeInterfaces {
     public static void main(String[] args) {
         A objA = new A();
         objA.metodo3();
         A.B objB = new A.B();
         objB.metodo1();
     }
}
```

En este caso, los métodos de la clase B sólo pueden acceder a los miembros estáticos de la clase A, dado que no son creados con referencia a un objeto de A, por lo que "metodo2" requiere ser estático para poder ser llamado desde "metodo2". La sintaxis de creación de un objeto B desde fuera de la clase A, en "main", también cambia.

En el caso de C++ y C#, los objetos de las clases inner no conservan automáticamente una referencia a un objeto de la clase outer, por lo que se les puede considerar clases inner estáticas por defecto, por lo que no requieren ni permiten su declaración utilizando el modificador "static".

Finalmente las clases inner pueden declararse con los mismos modificadores de acceso de los demás miembros. En el ejemplo anterior, si la clase "B" hubiera sido declarada como "private", no hubiera podido crearse ni declarar objetos de ésta fuera de los métodos de la clase "A".

Los Conflictos de Nombres

Cuando un miembro de la clase inner coincide en nombre con uno de la clase outer, es necesario utilizar una sintaxis especial que permite resolver a qué miembro se está llamando. La siguiente sintaxis corresponde a la forma de referirse a un miembro de una clase outer desde dentro de una clase inner:

C++:

```
<nombre de la clase outer>::<nombre del miembro>
```

Java: Desde una inner de instancia:

```
<nombre de la clase outer>.this.<nombre del miembro>
```

Desde una inner estática:

```
<nombre de la clase outer>.<nombre del miembro>
```

C#:

```
<nombre de la clase outer>.<nombre del miembro>
```

El siguiente programa en Java muestra este caso para una clase inner de instancia.

```
class Outer {
    class Inner {
        void m() {
            System.out.println("Llamada a Inner.m");
            Outer.this.m();
        }
    }
    void m() {
        System.out.println("Llamada a Outer.m");
    }
    void prueba() {
        Inner objInner = new Inner();
        objInner.m();
    }
}
public class PruebaDeInterfaces {
     public static void main(String[] args) {
        Outer objOuter = new Outer();
        objOuter.prueba();
     }
}
```

Note en el ejemplo anterior, que la expresión "Outer.this" es la forma en que se referencia al objeto de la clase outer en base al que se creó el objeto de la clase inner.

Las Clases Anidadas Anónimas

Java ofrece la capacidad de crear clases anidadas anónimas. Estas clases anónimas se instancian en la misma expresión que las define. El siguiente programa en Java muestra el uso de este tipo de clase.

```
class Base {
    public void Imprimir() { System.out.println("Base.Imprimir"); }
}
class Principal {
    public static void main(String args[]) {
        Base obj = new Base() { // Aquí comienza la definición de la clase anónima
            public void Imprimir() {
                   System.out.println(this.getClass().getName()+".Imprimir");
            }
        }; // Aquí termina la definición de la clase anónima
        obj.Imprimir();
    }
}
```

En el programa anterior, la clase inner anónima está definida entre los dos corchetes que siguen a la expresión de creación "new Base( )". Esta clase anónima hereda de la clase "Base" y su nombre no es requerido, debido a que el programa sólo requiere crear un objeto de dicha clase. El objeto de esta clase anónima es manejado siempre utilizando una variable del tipo de la clase base. El siguiente ejemplo en Java define una clase anónima que implementa una interfaz.

```
interface UnaInterface { void Imprimir(); }
class Principal {
    public static void main(String args[]) {
        UnaInterface obj = new UnaInterface() {
```

```
            public void Imprimir() {
                   System.out.println(this.getClass().getName()+".Imprimir");
            }
         };
         obj.Imprimir();
      }
   }
```

El código del método "Imprimir" muestra en consola el nombre interno que el compilador de Java le asigna a la clase anónima definida, en este caso "Principal$1". Si se revisa el directorio donde se generan los archivos compilados de Java para este programa se verá el archivo "Principal$1.class" correspondiente a esta clase inner.

Las clases anónimas son útiles cuando:

- Sólo se desea crear objetos de esta clase en una sola parte del programa, para extender una clase base o implementar una interfaz.
- La implementación de la clase anónima es relativamente pequeña.

La Reflexión

Cuando se trabaja con árboles de clases con método polimórficos o con interfaces es común tratar los objetos con referencias a las clases base o al tipo de las interfaces. A la asignación de un objeto de una clase derivada a una variable de una clase base se le conoce como "up-cast" o "widening". Este tipo de asignaciones no requiere de una operación de cast explícita y es segura, dado que un objeto de un tipo derivado "siempre" pueden tratarse como un objeto de un tipo base.La operación contraria no es segura, es decir, asignar una referencia desde una variable de un tipo base a una de un tipo derivado "no siempre" es correcta, dado que siempre es posible que se esté referenciando a un objeto que no puede ser tratado como el tipo de la variable destino. Este tipo de operación requiere de un cast explícito, y se conoce como "down-cast" o "narrowing". El siguiente código es un ejemplo de esto:

```
ClaseBase ref1 = new ClaseDerivada( ); // Up-cast o widening, siempre es seguro

... // otras líneas de código

ClaseDerivada ref2 = ref1; // Down-cast o narrowing, error de compilación,
                           // se requiere una operación "cast"
ClaseDerivada ref3 = (ClaseDerivada)ref1; // Esto si compila
```

En el código anterior, la última línea puede ocasionar un error al ser ejecutada, si la variable "ref1" referenciara a un objeto de la ClaseBase, producto de una operación de asignación ejecutada en alguna de las "otras líneas de código".

Dado que este tipo de operaciones cast son requeridas en algunas ocasiones, suele ser necesario contar con algún mecanismo para poder verificar si el objeto referenciado o apuntado por una variable puede o no ser tratado como un tipo determinado. Éste es un ejemplo en donde la técnica conocida como "reflexión" es útil.

Definición y Uso

Los lenguajes de programación que implementan la reflexión guardan información adicional en los objetos que son creados durante la ejecución de un programa de forma que, en tiempo de ejecución, sea posible obtener información sobre el tipo con que fue creado dicho objeto. La identificación de tipos en tiempo de ejecución (o RTTI por sus siglas en inglés) es sólo una de las capacidades que ofrece la

reflexión. La reflexión permite conocer el árbol de herencia de un tipo de dato, los modificadores con que fue definido, los miembros que incluye así como información sobre cada uno de estos miembros, entre otros datos.

Algunos ejemplos del uso de la reflexión son:

- Verificación de errores en tiempo de ejecución. Al realizar un cast de una referencia de un tipo base a un tipo derivado, la reflexión puede servir para averiguar si dicho objeto puede ser o no interpretado como tal o cual tipo, es decir, si su clase es o hereda del tipo destino. Si el objeto no puede ser tratado como del tipo destino, se produce un error que puede ser interceptado y tratado por el programa.

- Entornos de desarrollo con componentes. Estos entornos requieren exponer al programador los elementos de dichos componentes, así como su descripción, tipo, parámetros, etc. Se pueden instalar nuevos componentes y el sistema deberá poder reconocerlos automáticamente, siempre que cumplan con el estándar pedido por el entorno, esto es, que dichos componentes expongan la interfaz requerida.

- Programas orientados a dar servicios. Estos programas trabajan con otros programas que deben cumplir con ofrecer cierta interfaz. El programa servidor determina, en tiempo de ejecución, si otro programa cumple la interfaz necesaria para dar cierto servicio y si lo hace, puede trabajar con él. Un ejemplo de un programa de servicio es la misma arquitectura de un sistema operativo, donde los programas ejecutables y las librerías deben exponer cierta interfaz para que el sistema operativo pueda ejecutarlos. Otro ejemplo son los servidores Web y en general, cualquier servidor distribuido.

La reflexión no es la única técnica posible para éstos y otros casos donde un sistema requiera conocimiento de si mismo y de su entorno para adaptarse, pero ofrece la ventaja de ser simple y uniforme.

El tema de la reflexión es amplio, por lo que la siguiente sección sólo abarca lo que corresponde a RTTI.

RTTI

Uno de los aspectos de la reflexión es la identificación de tipos en tiempo de ejecución o RTTI. Esta consiste en determinar si un objeto puede ser manejado como un tipo de dato determinado.

C++ expone una implementación limitada de RTTI y requiere que los programas que la utilizan sean compilados con opciones especiales del compilador. El siguiente programa en C++ utiliza esta técnica.

```
#include <iostream.h>
#include <typeinfo.h>
class Base {
    public: virtual void Imprimir() { cout << "Base::Imprimir" << endl; }
};
class Derivada : public Base {
    public: virtual void Imprimir() { cout << "Derivada::Imprimir" << endl; }
};
void Identificar(Base* pObj) {
    Derivada* pDer = dynamic_cast<Derivada*>(pObj);
    if(pDer != 0)
        cout << "Objeto 'Derivada'" << endl;
    else
        cout << "Objeto 'Base'" << endl;
    const type_info& ti = typeid(*pObj);
    cout << "typeinfo: name=" << ti.name() << ", raw_name=" << ti.raw_name() << endl;
}

void main() {
```

```
    Base* pBase = new Base();
    Identificar(pBase);
    Derivada* pDer = new Derivada();
    Identificar(pDer);
}
```

El operador "tipeid" retorna una referencia de tipo "const type_info &", que es un objeto que contiene información acerca del tipo de objeto pasado al operador.

El operador "dynamic_cast" permite obtener un puntero de un tipo derivado, pasándole como parámetro un tipo base y el puntero original. Sin embargo, requiere que el tipo base contenga por lo menos un método "virtual". Si el puntero pasado como parámetro no apunta a un objeto que puede interpretarse como el tipo indicado entre los símbolos "< >", el operador retorna cero "0".

Java utiliza el operador "instanceof" para determinar si el objeto referenciado por una variable puede o no ser manejado como un tipo de dato determinado. El siguiente programa muestra su uso.

```
class Base { }
class Derivada extends Base { }

class PruebaDeInstanceof {
    static void Identificar( Base obj ) {
        if ( obj instanceof Derivada ) System.out.println("Objeto 'Derivada'");
        else System.out.println("Objeto 'Base'");
    }
    public static void main( String args[] ) {
        Identificar( new Base( ) );
        Identificar( new Derivada( ) );
    }
}
```

Si un objeto puede ser tratado como un tipo de dato determinado, la asignación de ésta, utilizando una operación de cast explícita, a una variable del tipo destino es segura y no arrojará error durante su ejecución.

C# utiliza el operador "is" en lugar del operador "instanceof" de Java. El siguiente programa muestra su uso.

```
using System;
class Base { }
class Derivada : Base { }
class PruebaDeIs {
    static void Identificar(Base obj) {
        if ( obj  is  Derivada )  Console.WriteLine("Objeto 'Derivada'");
        else Console.WriteLine("Objeto 'Base'");
    }
    public static void Main() {
        Identificar(new Base());
        Identificar(new Derivada());
    }
```