Capítulo

1

Introducción a Java y C#

Introducción a Java

Breve Historia

1991: Sun Microsystems, dentro de un proyecto interno (Green) crea un lenguaje Oak (autor: James Gosling), destinado a lo que ellos pensaban sería el próximo gran desarrollo: dispositivos electrónicos de consumo inteligentes. Lenguaje basado en C++.

1993: La WWW explota en popularidad. SUN encuentra nuevas posibilidades para su lenguaje, en el desarrollo de páginas WEB con contenido dinámico. Se revitaliza el proyecto.

1995: Se anuncia JAVA. Generó interés inmediato debido al ya existente interés sobre la WWW.

La Concepción de Java

Mientras que otros lenguajes fueron concebidos para facilitar la investigación científica, la enseñanza académica o para aplicaciones específicas, JAVA tuvo una motivación comercial. Esto significaba que debía cumplir con objetivos tales como:

1. Ser fácil de aprender y de utilizar. Un lenguaje que si bien está orientado a objetos, debe evitar todas las sofisticaciones que un lenguaje como C++ implementa y lo hacen difícil de aprender y de utilizar.

2. Tener un ciclo de desarrollo rápido. El lenguaje deberá proporcionar herramientas para todos los trabajos más comunes durante la programación, de forma que el programador sólo se concentre en agregar nueva funcionalidad. Los ciclos de prueba y depuración también deben ser rápidos.

3. Ser confiable. El lenguaje deberá llevar al desarrollador a programar de forma tal que el programa final sea estable (sin caídas). Además, el lenguaje deberá prevenir que cualquier acción del programador afecte al sistema operativo inestabilizándolo.

4. Ser seguro. Al estar orientado al desarrollo para Internet, el lenguaje deberá brindar características de seguridad tanto en la protección del programa distribuido (contra procesos de ingeniería inversa) como en la facilidad de poder establecer permisos de acceso.

5. Ser portable. Java debía de ser independiente de la plataforma de ejecución (concepto de Arquitectura Neutra). Un programa Java debería generarse una única vez y poder ejecutarse en cualquier plataforma (hardware + software) que cuente con un Intérprete Java.

El producto final de esta concepción, JAVA, es un lenguaje de programación orientado a objetos. Las piezas de programación se denominan clases.

Java esta formado por librerías de clases. El conjunto de librerías estándar es extenso y cubren las actividades más comunes durante el desarrollo de programas de modo que el programador pueda concentrarse sólo en desarrollar la nueva funcionalidad.

Las librerías de clases de Java pueden agruparse en 3 grupos:

1. Estándar. Desarrolladas y distribuidas libremente por SUN Microsystems conjuntamente con las herramientas básicas de desarrollo del lenguaje.

2. Desarrolladas por terceros. El programador necesitará comprarlas para poder utilizarlas.

3. Desarrolladas por el mismo programador. Java permite crear nuevas librerías de forma que pueda ser utilizada por muchos programas desarrollados por el programador.

El punto de referencia respecto a la sintaxis del lenguaje fue C++. En este sentido, ambos lenguajes comparten muchos conceptos como, por ejemplo, la programación multitarea.

Entorno de Trabajo de Java

Aunque el desarrollo de un programa en Java puede realizarse utilizando únicamente el Kit de desarrollo proporcionado por SUN llamado JDK (Java Developer Kit), existen muchos entornos de desarrollo integrados (IDE) que facilitan este proceso. Los más conocidos en nuestro medio son:

1. JBuilder — IDE de Borland
2. Borland C++ 5.0 — IDE para C++ de Borland con extensión para Java
3. Visual J++ — IDE de Microsoft.
4. Visual Age for Java — IDE de IBM.

Existe JDK's apropiados para trabajar en los principales sistemas operativos. El JDK consiste principalmente en:

1. Las librerías que implementan el lenguaje
2. El resto de librerías estándar
3. Un compilador
4. Uno o más intérpretes
5. Programas utilitarios adicionales

Ciclo de Desarrollo

El desarrollo de un programa en JAVA pasa por 5 fases:

1. Las 2 primeras corresponden a la creación del programa

    1.1. Edición. Las fuentes de Java son archivos de texto con extensión JAVA. Un programa en Java puede escribirse utilizando cualquier editor de texto. Si el archivo contiene una clase pública, su nombre **debe** corresponder **exactamente** al nombre de esta clase.

    1.2. Compilación. Se utiliza un compilador para JAVA (javac.exe para Borland, jvc.exe para Microsoft). Este programa recibe como entrada el archivo fuente (*.java) y genera un archivo compilado con el mismo nombre que el anterior, pero con extensión CLASS (ejemplo: Hola.class). Este archivo compilado contiene juegos de instrucciones llamadas BYTECODE, no código maquina, por lo que no puede ser ejecutado directamente por el sistema operativo.

2. Las 3 últimas corresponden a la ejecución del programa generado

    2.1. Carga de la Clase. Lo realiza un programa *Cargador de Clases* que se encarga de colocar en memoria el bytecode del archivo a procesar.

    2.2. Verificación del BYTECODE. Lo realiza un programa *Verificador de BYTECODE* que se encarga de revisar que todo el BYTECODE leído y cargado a memoria sea válido y que no viole las restricciones de seguridad de JAVA.

    2.3. Interpretación. Lo realiza un programa *Intérprete*, el cual lee el BYTECODE y lo convierte a instrucciones en lenguaje máquina, de forma que pueda ser ejecutado.

Tipos de Programas en Java

Existen 2 tipos de programas que se pueden generar con JAVA:

- Aplicaciones
- Applets

Las aplicaciones son programas que normalmente se almacenan y ejecutan en la propia máquina del usuario. Un programa de JAVA se carga utilizando un intérprete de JAVA (como se explicó líneas arriba).

El applet es un programa hecho para correr dentro de un browser de WEB (como el Netscape, Explorer, etc.) el cual, internamente, es el responsable de llamar al intérprete. Un applet comunmente es cargado desde una computadora remota.

Un applet también puede ser visto desde un visor de applets (llamado **appletviewer.exe** en el JDK, **jview.exe** en Visual J++). Este programa es el browser mínimo (pues sólo reconoce el tag *applet*) necesario para probar un applet y recibe como entrada un HTM.

Nota: JavaScript es un lenguaje independiente y diferente al JAVA, además de tener propósitos diferentes.

Compiladores, Intérpretes Puros e Intérpretes JIT

Cuando un archivo fuente Java es compilado, lo que se genera es BYTECODE, que no es otra cosa que una serie de instrucciones "genéricas" (su formato y significado no está amarrado a ninguna arquitectura), que se corresponden con las instrucciones dadas en el archivo fuente. Sin embargo, a diferencia del archivo fuente que tiene un formato de texto, el BYTECODE es una secuencia binaria, haciendo que su lectura y velocidad de ejecución sea mucho más eficiente.

Java es un lenguaje interpretado. Esto quiere decir que las instrucciones de un programa en Java son traducidas por otro programa que hace de intermediario (Intérprete) entre el lenguaje Java y el entorno de ejecución (hardware + sistema operativo). Los intérpretes leen una instrucción Java y ejecutan su equivalente en instrucciones que pueden ser ejecutadas directamente por el computador (código nativo).

Al comienzo, del lado del cliente sólo se tenía disponibles programas intérpretes. Sin embargo, el costo adicional que el proceso de interpretación significa hace que un programa interpretado sea comparativamente mucho más lento que un programa compilado. Debido a esto se desarrollaron Compiladores Reales, de BYTECODE a código nativo, orientados a programas en donde la máquina del cliente es conocida. El programa generado, al igual que cualquier programa compilado a código nativo, es dependiente de la plataforma de ejecución del cliente.

Dado que la interpretación de un código cuyas instrucciones no están amarradas a ninguna arquitectura (arquitectura neutra) es esencial para cumplir con el objetivo de portabilidad que se buscaba con el lenguaje, y que Java fue pensado para funcionar como lenguaje para desarrollo de aplicaciones en Internet, un ejecutable de Java tendría que ser transferido al computador en donde se desea ver una página HTML que contenga un applet de Java. Dado que un ejecutable Java es mucho más extenso que un compilado a BYTECODE, se crea un conflicto entre la velocidad de ejecución y la velocidad de carga inicial de una pagina HTML que contenga un programa applet de Java.

Una solución parcial a este problema lo resuelven los llamados Intérpretes JIT (Just-In-Time). Un JIT funciona de la misma manera que un intérprete normal la primera vez que ejecuta cada parte de un BYTECODE, pero genera a la par un cuasi-código-nativo de dichas instrucciones de forma que la siguiente vez que se intente ejecutar las mismas instrucciones se utilizará el nuevo código generado. Esto hace un poco más lenta la ejecución inicial pero hace rápida la carga de la página HTML que contiene el applet (dado que lo que se transfiere es BYTECODE) y acelera las sucesivas ejecuciones del programa.

Paralelo entre Java y C++

Algunas diferencias notorias entre la programación bajo C++ para Windows y JAVA se muestran en la tabla 1.1

Tabla 1.1. Diferencias entre Java y C++

C++	JAVA
Los programas utilizan programación estructurada y/o programación orientada a objetos.	Toda la programación es orientada a objetos.
La implementación de clases sigue el esquema del bosque de árboles.	La implementación de clases sigue el esquema del árbol único.
La implementación de los métodos de una clase puede estar dentro de ella o fuera de su declaración.	Toda la implementación de una clase debe estar dentro de la clase.
Herencia de clases múltiple.	Herencia de clases simple + implementación de interfaces.
Los fuentes están formados por archivos cabecera y un archivo de implementación.	Toda la implementación se hace en un sólo archivo.
Se compila generando CODIGO NATIVO.	Se compila generando BYTECODE.
La compilación genera archivos EXE, LIB, DLL, OCX, etc.	La compilación genera archivos CLASS.
EXISTEN PUNTEROS.	NO EXISTEN PUNTEROS.
Existen clases y estructuras, que permiten almacenar datos de diversa índole.	Sólo existen clases.
En Windows (y en otros sistemas operativos donde se trabaje bajo el esquema de ventanas) la programación se maneja mediante mensajes. Se trabaja en base a funciones de procesamiento de mensajes.	Programación manejada por eventos. Se trabaja mediante métodos que se disparan en respuesta a una acción del usuario. El trabajo interno con los mensajes es ocultado por el lenguaje.
Se pueden sobrecargar los operadores para una clase.	No existe sobrecarga de operadores.
La liberación de la memoria solicitada durante el programa es responsabilidad del mismo programa.	La liberación de la memoria solicitada durante el programa es responsabilidad del propio lenguaje JAVA, mediante el *Garbage Collector*.

Algunas semejanzas notorias entre la programación bajo C++ para Windows y JAVA se muestran en la tabla 1.2

Tabla 1.2. Semejanzas entre Java y C++

C++ y JAVA
Ambas utilizan la misma sintaxis (salvo mínimas diferencias) al escribir expresiones, sentencias, funciones, declarar variables, declarar clases, etc.
Ambos utilizan los mismos operadores aritméticos y lógicos, con las mismas reglas de precedencia y agrupación.
La implementación de una clase (la herencia, el polimorfismo, etc.) son muy similares.
El juego de palabras reservadas y su uso es muy similar, por ejemplo, los objetos pueden hacer referencia a si mismos mediante la palabra clave *this*.
Tienen la misma implementación de las estructuras de control (if, if/else, while, do/while, switch, for).
Los modificadores de ámbito de acceso son muy similares en su uso y reglas (public, private, protected, etc.).

Introducción a la Programación

Las siguientes secciones dan una introducción a la programación en Java y C# mediante ejemplos de programas básicos.

Aplicaciones Java: Ejemplo 1

Para mostrar los detalles más básicos sobre la programación en Java analizaremos una aplicación Java simple que muestre un mensaje.

Los fuentes de un programa en Java son archivos de texto con extensión JAVA, por lo que el siguiente código puede ingresarse y guardarse con cualquier editor de texto disponible.

```
// Archivo: Aplicacion1.java
/* Descripción:
       El programa muestra un mensaje en pantalla
*/
public class Aplicacion1 {
       public static void main( String args[ ] ) {
              System.out.println( "Aplicacion1 a Java" );
       }
}
```

Los comentarios en un archivo fuente de Java siguen la misma sintaxis y reglas que C/C++.

Al igual que C/C++, Java especifica un juego de palabras reservadas. Las palabras *class, public, void,* etc. son reservadas por Java y no deben de utilizarse para definir identificadores. Todas las palabras reservadas de Java están en minúscula.

En el ejemplo, las palabras Aplicacion1, main, System, out, println, String, etc. son identificadores.

La sintaxis para especificar literales (numéricos, carácteres y cadenas de carácteres) es la misma que C/C++. En el ejemplo anterior se especifica un literal tipo cadena.

Las sentencias en Java siguen las mismas reglas que C/C++. Una sentencia siempre debe finalizar con un ';'; sin embargo, note que Java no requiere que se coloque un ';' al final de la implementación de la clase.

La implementación de una clase en Java es, en términos generales, muy similar a C++.

A continuación analizaremos los aspectos de este programa referentes a la programación en Java.

Compilación y Ejecución de una Aplicación Java

Para compilar un fuente JAVA se puede utilizar el compilador correspondiente al IDE o entorno de trabajo con que se cuente, por ejemplo:

- JVC.EXE Correspondiente al IDE de Microsoft
- JAVAC.EXE Correspondiente al IDE de Borland y al SDK de Sun.

En el caso de contar con un IDE, normalmente es más sencillo dejar que éste se encargue de la tarea de compilación en vez de realizarla nosotros mismos desde la línea de comandos; sin embargo, por motivos didácticos, realizaremos esto último.

En nuestro ejemplo, para compilar el archivo Aplicacion1.java se puede utilizar la siguiente instrucción:

```
C:\>javac.exe Aplicacion1.java
```

Esta instrucción utiliza el compilador de Borland (parte de su IDE para Java, JBuilder) para compilar el archivo editado. Si la compilación fue exitosa, se genera el correspondiente archivo: *Aplicacion1.class*.

Para ejecutar un archivo CLASS obtenido de la compilación de un archivo fuente JAVA necesitaremos correr el intérprete de Java y pasarle como parámetro nuestro archivo compilado. Nuevamente, se puede utilizar el intérprete correspondiente al IDE o entorno de trabajo con que se cuente, por ejemplo:

- JVIEW.EXE y WJVIEW.EXE Correspondiente al IDE de Microsoft
- JAVA.EXE y APPLETVIEWER.EXE Correspondiente al IDE de Borland y al SDK de Sun.

En nuestro ejemplo, para ejecutar el archivo *Aplicacion1.class* se puede utilizar la siguiente instrucción:

```
C:\>java.exe Aplicacion1
```

Al ejecutar esta instrucción, se carga y ejecuta la aplicación Java contenida en el archivo *Aplicacion1.class*, mostrándose el mensaje "Aplicacion1 a Java" en pantalla.

Note que en los ejemplos anteriores se asume que tanto el archivo fuente, el compilador, el archivo compilado y el intérprete se encuentran en la misma carpeta, la raíz de la unidad C, o bien en carpetas donde el sistema operativo sea capaz de encontrarlos (para Windows, los directorios del sistema y las carpetas configuradas en la variable de entorno PATH). Si éste no fuese el caso, se tendría que especificar la ruta completa de estos archivos en la instrucción. Para más detalles acerca de las formas en que se puede compilar y ejecutar un programa Java y los posibles errores que pueden ocurrir, se puede consultar el tutorial de Java en Internet:

http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html

Note que el ejemplo anterior corresponde a una aplicación de consola. El archivo implementado contiene un método "main" que es el punto de entrada que utilizará el intérprete cuando lo que se ejecuta es una aplicación Java. El intérprete llamará a éste método y éste será el encargado de ejecutar todas las acciones de la aplicación. A las clases para una aplicación Java que contengan el método "main" se les podrá utilizar como punto de inicio de la aplicación. Debido a esto, a estas clases se les llama *clases ejecutables*. El intérprete no crea un objeto *Aplicacion1*, su única responsabilidad es llamar al método estático "main" de esta clase (lo que, al igual que C/C++, no requiere que se cree el objeto para ser llamado). El método "main" será el encargado de crear los objetos necesarios para realizar las tareas de la aplicación.

Aplicaciones Java: Ejemplo 2

El siguiente ejemplo agrega utiliza algunas características adicionales comunes a los programas en Java.

```
// Archivo: Aplicacion2.java
// Descripción:
//     Se utiliza una caja de dialogo para mostrar un mensaje.

import javax.swing.JOptionPane;

class Aplicacion2 {
       public static void main( String args[ ] ) {
              String sMensaje;

              sMensaje = "Aplicacion1 a java!!!\n";
```

```
                sMensaje += "Los argumentos pasados en la línea de comando son:\n";
                for( int i = 0; i < args.length; i++ )
                        sMensaje += args[i] + "\n";
                sMensaje += '\n';

                double dResultado = RaizCuadrada( 10.5 );
                sMensaje += "La raíz cuadrada de 10.5 es " + dResultado;

                int iDato = 40;
                long lResultado = (long)Multiplicar( 24, iDato );
                sMensaje += "\nLa multiplicación de 24 * " + iDato + " es " +
lResultado;
                JOptionPane.showMessageDialog( null, sMensaje );

                System.exit( 0 );
        }

        public static double RaizCuadrada( double dValor ) {
                double dRaiz;
                dRaiz = Math.sqrt( dValor );
                return dRaiz;
        }

        public static long Multiplicar( int iValor1, int iValor2 ) {
                return iValor1 * iValor2;
        }
}
```

La aplicación anterior utiliza una caja de diálogo para mostrar un mensaje, en lugar de hacerlo por la salida estándar, la consola.

Note que los operadores, las estructuras de control, la declaración de variables y su inicialización, la declaración de los métodos y sus parámetros, la llamada a métodos y el paso de parámetros, el retorno de valores de un método y las operaciones de tipo *cast* para conversión entre tipos primitivos son iguales a C++.

El ejemplo utiliza una variable de tipo arreglo. Los detalles acerca del uso de arreglos se verán más adelante. También utiliza la clase JOptionPane que pertenece a la librería “javax.swing”, por lo que una sentencia “import” al inicio del archivo debe de ser agregada indicándole al compilador dónde se define la clase que estoy utilizando. Las librerías en Java se denominan “paquetes”.

La clase JOptionPane permite mostrar ventanas con mensajes y botones de selección. Siempre que se utilicen elementos gráficos (como las ventanas) se deberá finalizar una aplicación llamando al método *System.exit,* el cual recibe como parámetro un número entero que representa el valor de retorno de la aplicación. Estos valores de retorno son comunmente utilizados en archivos de procesamiento por lotes, como los archivos BAT de Windows, de la misma forma que el valor de retorno de la función “main” de un programa en C/C++.

Introducción a las Variables

La declaración de variables en Java es similar a C++. Por su tipo de datos las variables se clasifican en:

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

- Variables a datos de tipo objeto, también llamadas variables referencia o simplemente referencias.

Toda variable debe de ser inicializada antes de su uso. Por ejemplo, el siguiente código arrojará un error al momento de compilarse:

```
String sTexto;
sTexto = sTexto + “Hola ”;
sTexto += “mundo”;
```

La segunda línea del código falla al tratar de concatenar el objeto referenciado por sTexto con el objeto literal "Hola ". Esto se debe a que sTexto no se inicializó previamente indicando a qué objeto iba a referenciar. Este código puede corregirse de la siguiente forma:

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

Convención de Nombres

Si bien los nombres de los paquetes, clases, métodos, datos y etiquetas pueden tener cualquier combinación de mayúsculas y minúsculas, se suele usar la siguiente convención:

- **Paquetes:** Todas las letras en minúscula. Ejemplo: java.applet
- **Clases:** Una serie de palabras juntas (sin espacios), con la primera letra de cada palabra en mayúscula y el resto en minúscula. Ejemplo:

```
class DatosPersona { ... }
```

- **Métodos:** Una serie de palabras juntas (sin espacios), la primera palabra suele representar una acción con todas sus letras en minúscula, el resto de palabras (si las hay) describen "sobre qué" se realiza la acción, con la primera letra de cada una en mayúscula y el resto en minúscula. Ejemplo:

```
public void dibujarTrianguloEquilatero( int Lado ) { ... }
```

Las palabras reservadas siempre son en minúscula. Cabe recalcar que JAVA es sensitivo a las mayúsculas y minúsculas, lo que significa que si colocamos una palabra reservada con alguna letra en mayúscula el compilador de Java no la reconocerá y arrojará un error.

Conclusiones Preliminares

Por lo revisado en los 2 simples códigos de ejemplo anteriores, todo programa en Java consiste básicamente en la definición de una o más clases. Nada está fuera de las clases.

Aunque no se vio en los ejemplos, cada archivo Java puede contener la implementación de más de una clase, sin embargo sólo una de ellas puede ser pública (y su nombre debe corresponder exactamente al del archivo en donde está definida), esto es, declarada como *public*. El uso de estos modificadores de acceso se verá más adelante.

Elementos de la Programación

Identificadores

Todo identificador es un conjunto de carácteres, sin espacios, que no pueden comenzar con un dígito y que pueden contener:

- Letras (a-z, A-Z).
- Dígitos (0-9).
- Los carácteres '$' y '_'.

El carácter '$' debe evitarse dado que es usado por el compilador para representar internamente los identificadores.

Al igual que C/C++, Java es sensitivo a las mayúsculas y minúsculas en el nombre de los identificadores.

Operadores

Operadores Aritméticos

Son operadores binarios (actúan sobre 2 operandos). El tipo del valor resultado depende del tipo de dato de los operandos, siguiendo las reglas de promoción de tipos.

+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo (sólo con operandos enteros)

Operadores Relacionales

Son operadores binarios. El resultado es un valor booleano.

==	Igualdad
!=	Diferencia
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Operadores Lógicos

Trabajan sobre operandos booleanos o expresiones que al ser evaluadas devuelvan un booleano. El resultado es un booleano. Siguen las reglas de evaluación de circuito corto.

&&	AND	Operador Binario
||	OR	Operador Binario
!	NOT	Operador Unario

Operadores Lógicos Booleanos

Trabajan a nivel de bits sobre operandos de tipo primitivo e integral. Si los operandos son de diferente tipo (por ejemplo int y byte) el resultado es de un tipo promovido (byte se promociona a int y el resultado es de tipo int). Cuando los operandos son booleanos, estos operadores se comportan como los Operadores Lógicos. Siguen las reglas de evaluación de circuito largo.

&	AND
		OR inclusivo
^	OR exclusivo

Operadores de Incremento y Decremento

Son operadores unarios (actúan sobre un sólo operando). El valor resultado es del mismo tipo del operando. Puede anteceder o preceder al operando. Cuando anteceden, se evalúan antes que la expresión de donde forma parte. Cuando están precediendo, se evalúan después que la expresión de donde forman parte.

++	Incrementa en uno el operando
--	Decrementa en uno el operando

Al igual que en C++, en una expresión con varios operadores, se aplica la misma regla de precedencia. Cuando se utilizan varios operandos con iguales o distintos operadores, se aplica la misma regla de agrupación de C++.

Tipos de Datos Primitivos

La tabla 1.3 muestra los tipos de datos primitivos en Java, el espacio de memoria que ocupan y el rango de valores que pueden tomar.

Tabla 1.3. Tipos de datos primitivos

Nombre	Tamaño en bits	Valores
boolean	1	**true** o **false**
char	16	'\u0000' a '\uFFFF'
byte	8	Igual que C++
short	16	Igual que C++
int	32	Igual que C++
long	64	Igual que C++
float	32	Igual que C++
double	64	Igual que C++

Note que Java utiliza 2 bytes para representar un carácter. Esto se debe a que Java maneja carácteres siguiendo el estándar UNICODE.

Cuando en una expresión o una sentencia de asignación se mezclan datos primitivos de distinto tipo, cada evaluación de un *operador* requiere una homogenización de sus *operandos*, lo que se realiza siguiendo las reglas de *Promoción de Tipos* que se muestran en la tabla 1.4. Estas reglas de promoción se realizan automáticamente dado que no causan pérdida de información. Sin embargo, una conversión puede ser *forzada* mediante una operación *cast*, por ejemplo:

```
long x = 10;
int y = (int)x;
```

Tabla 1.4. Reglas de promoción de tipos de datos

Tipo primitivo	Puede ser promovido a
double	Ningún tipo, no existe ningún dato primitivo más largo
float	double
long	float o double
int	long, float o double
char	int, long, float o double
short	int, long, float o double
byte	short, int, long, float o double
bolean	Ningún tipo, los booleanos no son considerados números

Al igual que C/C++, este tipo de operaciones puede significar una pérdida de información.

Estructuras de Control

Existen las mismas estructuras de control que en C++, con la misma sintaxis y las reglas de evaluación. Éstas son:

- if / else
- switch
- for
- while y "do / while"

Dentro de los bucles (for, while, do / while) se pueden utilizar los controladores de flujo:

- continue
- continue <nombre de etiqueta>

- break
- break <nombre de etiqueta>

Nótese que a diferencia de C++, existen versiones con etiqueta para cada controlador. El detalle de estas características de Java escapa del alcance del curso.

Constantes Literales

Se siguen las mismas reglas que C++. Los sufijos para los literales numéricos, al igual que C++, modifican el tipo por defecto de éstos. La tabla 1.5 muestra ejemplo de declaración de constantes para diferentes tipos de datos.

Los literales de tipo cadena aceptan los caracteres de escape de C: "\n", "\t", "\r", "\\", "\"".

Tabla 1.5. Ejemplos de constantes literales

Literal	Tipo de Dato
178	int
8864L	long
37.266	double
37.266D	double
87.363F	float
26.77e3	double
' c '	char
true	boolean
false	boolean

El API de Java

Java cuenta con una extensa gama de clases predefinidas. A este conjunto de clases, agrupados en paquetes, se le conoce como el API de Java.

Los principales paquetes estándar del API de JAVA se muestran en la tabla 1.6:

Tabla 1.6. Paquetes estándar de Java

Nombre	Descripción
java.applet	Contiene la clase Applet junto con otras interfaces que permiten la creación y manejo de applets.
java.awt	*Java Abstract Window Toolkit*. Contiene los componentes básicos para crear y manejar aplicaciones con interfaz gráfica de usuario (GUI).
java.awt.datatransfer	*Java Data Transfer*. Permite transferir y recuperar datos hacia y desde el clipboard.
java.awt.event	Clases e interfaces que permiten manejar eventos para los componentes de este paquete.
java.awt.image	Clases e interfaces que permiten almacenar, recuperar y manipular imágenes.
java.awt.peer	Clases e interfaces de los componentes GUI con su implementación especifica para la plataforma actual de desarrollo. No debería ser utilizada por el programador pues rompería con la filosofía de JAVA de brindar aplicaciones que no dependan de la plataforma.
java.beans	Clases e interfaces que permiten crear componentes reutilizables tanto desde aplicaciones JAVA como NO-JAVA.
java.io	Permiten la lectura y escritura desde los dispositivos de entrada y salida estándar (teclado y pantalla por defecto).
java.lang	Clases e interfaces que forman parte del lenguaje JAVA.
java.lang.reflect	*Java Core Reflection Package*. Clases e interfaces que permiten averiguar en tiempo de ejecución que datos y métodos de una clase son accesibles.
java.net	*Java Network Package*. Permite comunicarse, utilizando alguno de los protocolos existentes, a través de la red con otros programas JAVA.
java.rmi java.rmi.dgc java.rmi.registry java.rmi.server	*Remote Method Invocation*. Clases e interfaces que permiten crear programas JAVA distribuidos. Usando RMI un programa puede llamar a un método de otro programa en la misma PC u otra que sea accesible a través de Internet.
java.security java.security.acl	Clases e interfaces que implementan la seguridad en JAVA. Permiten encriptar datos y controlar los privilegios de acceso de un programa en JAVA.

java.security.interfaces	
java.sql	Clases e interfaces que permiten la conexión y manejo de bases de datos.
java.text	Clases e interfaces para la manipulación de textos bajo diferentes formatos: representando números, fechas, etc. Provee muchas de las características de internacionalidad de JAVA.
java.util	Clases e interfaces utilitarios de JAVA. Permiten, entre otras cosas: - Manipulación de fecha y hora - Generación de números aleatorios - Almacenamiento y procesamiento de largas cantidades de datos - Tokenizar cadenas - Manejo de bits
java.util.zip java.util.jar	Clases e interfaces que permiten crear (y leer) archivos CLASS y otros archivos de recursos (imágenes, textos, etc.) en un sólo archivo comprimido de tipo *JAR* (*Java Archive File*).
javax.accessibility	Clases e interfaces que permiten agregar funcionalidad a un programa que facilite su interacción con personas discapacitadas.
javax.swing	Clases, interfaces y otros sub-paquetes que permiten utilizar el nuevo juego de componentes SWING. Reemplazan a muchos de los elementos proporcionados por los paquetes (y sub-paquetes bajo éstos) *java.applet* y *java.awt*.
org.omg.CORBA org.omg.CosNaming	Clases, interfaces y otros sub-paquetes que permiten desarrollar programas que puedan comunicarse con otros lenguajes.

Entrada y Salida de Datos

En los ejemplos que hemos visto hasta ahora sólo se ha realizado salida de datos por consola y mediante una caja de diálogo.

El ingreso de datos por consola es más complicado que la salida. No existe, como podría esperarse, un objeto *Sistem.in* que sirva para manejar la entrada estándar desde consola. En su lugar existen diversas clases especializadas para el ingreso por consola, dependiendo del formato con el que se espera recibir los datos y desde donde se desea recibirlos, desde consola o un archivo.

También es posible utilizar una caja de diálogo para ingresar datos. Por ejemplo:

```
String sValor = JOptionPane.showInputDialog( "Ingrese un cadena de texto" );
```

Mostrará una caja de diálogo que contendrá una etiqueta, un control de edición de texto y dos botones, OK y CANCEL. Esto permitirá ingresar un texto que, al presionarse OK, será devuelto por el método showInputDialog.

También es posible implementar ventanas más complejas que contengan muchos otros elementos GUI, áreas de dibujo, menús, barra de herramientas y todos los elementos comunes al trabajo con ventanas.

Una última opción es el desarrollo de applets, que son ventanas dentro de la ventana de un browser de Internet.

Algunas de estas técnicas se verán más adelante.

Introducción a C#

Primer Programa

El siguiente es un programa simple, en modo consola, que escribe un mensaje en pantalla.

```
using System;

class MainClass {
        public static void Main(string[ ] args) {
                Console.WriteLine("Hello World!");
        }
}
```

Este código se coloca dentro de un archivo de texto, comúnmente con extensión CS, por ejemplo "Introduccion.cs". Una vez grabado, se puede compilar desde una ventana de comandos utilizando el compilador de C#, el programa CSC.EXE instalado como parte de ..NET Framework SDK, de la siguiente forma:

```
csc Introduccion.cs
```

Al ejecutarse este comando se genera el archivo "Introduccion.exe", el cual contiene código MSIL, por lo que no debe considerarse como un ejecutable estándar de código nativo.

Todo el código, inclusive el punto de entrada Main, debe definirse dentro de clases u otras definiciones de tipos de datos, como se verá más adelante. Todos los tipos de datos, predefinidos o definidos por el programador, están organizados en espacios de nombres. Cuando se definen tipos sin especificar un espacio de nombres, se asume que forman parte del espacio de nombres global. Se verán ejemplos más adelante.

La directiva **using** especifica un espacio de nombres, **System** en el ejemplo, donde el compilador puede buscar los tipos de datos utilizados en el programa y definidos fuera del espacio de nombres actual. La clase **Console** pertenece al especio de nombres System, por lo que si no se declara éste último utilizando using, se debería colocar el "nombre completo de la clase", esto es:

```
System.Console.WriteLine("Hello World!");
```

El formato de definición de una clase, si bien muy similar a C++, presenta algunas diferencias claras:

- ✍ Todos los miembros de una clase, datos y funciones, se declaran con todos sus modificadores de manera individual, por lo que estos modificadores sólo afectan a dicho elemento.

- ✍ No se finaliza una declaración de clase con un símbolo ';'.

El formato general de declaración de un método es:

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

Los modificadores especifican características de los métodos, por ejemplo los modificadores de acceso public, private y protected. El uso de los modificadores se verá más adelante.

El método Main es el punto de entrada de todo programa ejecutable en C#. Existen varios formatos, según se desee retornar un valor o manejar los argumentos ingresados en la línea de comando. Este método se define como estático, con el modificador static, debido a que el intérprete de ..NET, el CLR, accederá a él sin instanciar

un objeto de la clase dentro de la cual se define. Dado que este acceso se realiza desde fuera de la clase, el método Main debe definirse como público, con el modificador public.

El cuerpo del método Main imprime un mensaje en una ventana de consola, utilizando el método estático WriteLine de la clase Console. Más adelante se verá el tema de la entrada y salida de datos.

Comparación con C++

C# esta basado en C++, por lo que comparte su misma sintaxis para la declaración de variables, métodos, sentencias, operadores, controles de flujo y tipos de datos, con algunas modificaciones que se indicarán a lo largo del presente documento.

La declaración de variables tiene, salvo algunas excepciones, la misma sintaxis de C++. A diferencia de C++, en C# no es posible declarar variables en un ámbito global, sólo en ámbito de clase (datos miembros) y en ámbito de funciones (variables locales).

La declaración de funciones tiene, salvo algunas excepciones, la misma sintaxis de C++. A diferencia de C++, en C# no es posible declarar funciones en un ámbito global (funciones globales), sólo en ámbito de clase (métodos).

En general, la sintaxis de las expresiones así como sus reglas de evaluación son las mismas de C++.

Los operadores y su regla de precedencia son muy similares a C++, con las siguientes modificaciones:

- Los operadores de acceso a los miembros de clases y estructuras, '->' y '::', se reemplazan por el operador '.' (punto).
- El operador '&' desaparece, tanto para obtener una dirección de memoria como para definir una variable tipo referencia al estilo de C++.
- El operador '*' para indireccionamiento desaparece.
- Se agregan los siguientes operadores:
  - checked y unchecked: Sirven para verificar errores de desbordamiento.
  - is: Sirve para verificar si un objeto de datos puede ser interpretado como un tipo de dato en particular.
- Los controles de flujo de C++ se utilizan en C# con la misma sintaxis pero con los siguientes cambios:
  - Las expresiones que evalúan una condición deben retornar un tipo de dato booleano.
  - El control de flujo switch soporta literales de texto.
  - Se define el control de flujo foreach y using.
  - Se define el bloque finally, como un bloque de ejecución obligatoria, para el manejo de excepciones.

Tipos de Datos

C#, al igual que todos los lenguajes de programación compatibles con ..NET, basan sus tipos de datos en el sistema de tipos comunes de ..NET o Common Type System (CTS). Este sistema incluye tanto tipos simples

(int, char, float, etc.) como complejos (como string y decimal). Todos estos tipos de datos son realmente clases, las que tienen métodos para acciones comunes, por ejemplo: conversión a texto, serialización, identificación del tipo en tiempo de ejecución, conversión a otros tipos, etc.

C# es un lenguaje fuertemente tipificado, a diferencia de C y C++. Por ejemplo, un tipo de dato booleano no se convertirá automáticamente a un entero, a menos que se indique explícitamente que se desea esa conversión, mediante una operación cast. También es posible especificar el comportamiento de un tipo de dato definido por el usuario, cuando se enfrenta a una conversión de tipos explícita e implícita.

Los objetos de datos en C#, variables y constantes, pueden estar almacenados en el stack o en el heap. Los objetos de datos del stack pueden ser primitivos y estructuras. Los objetos de datos del heap corresponden al resto de tipos de datos. A diferencia de C y C++, en C# es el lenguaje el que escoge la ubicación de un objeto de datos, en base a su tipo. A los tipos de datos en stack se les llama tipo valor, a los tipos de datos en heap se les llama tipo referencia.

El heap en C# funciona de manera diferente al de C y C++. En C#, el CLR crea y gestiona los objetos de datos durante la ejecución del programa, teniendo la responsabilidad de liberar la memoria no utilizada mediante un recolector de basura, el cual es un hilo que corre en paralelo al resto de hilos del programa pero con una prioridad baja, realizando la revisión de los objetos de datos del heap y liberando aquellos marcados como no-utilizados. A este heap se le llama heap gestionado.

La ubicación de almacenamiento de un tipo de dato implica cómo sus objetos se comportarán en una operación de asignación. Para los tipos valor se creará una copia del valor, teniéndose dos objetos de datos distintos almacenando el mismo valor. Para los tipos referencia se creará una copia de la referencia, teniéndose dos objetos de datos referenciando al mismo valor en la misma posición de memoria. En el siguiente ejemplo se crea una copia del valor de una variable entera, el cual es un tipo valor:

```
int a1 = 10;
int a2;
a2 = a1;
```

En este ejemplo, a1 y a2 son variables de tipo valor que almacenan en posiciones de memoria distintas, el mismo valor. En el siguiente ejemplo se crea una copia de la referencia de una variable cadena, la cual es un tipo referencia:

```
string s1 = "jose";
string s2 = s1;
```

En este ejemplo, s1 y s2 son variables de tipo referencia, que refieren al mismo objeto de datos en memoria, el cual almacena el literal “jose”.

Luego, para datos primitivos o complejos pero de poco tamaño, es más eficiente trabajarlos en el stack, como tipos valor, dado que se evita la sobrecarga que implica la creación y manejo de objetos de datos en el heap. Por ejemplo, no es deseable tener que crear dinámicamente cada entero que se utilice en un programa. Como contraparte, para datos predefinidos complejos y otros definidos por el programador, en donde se almacenan un número considerable de datos, es más eficiente crearlos en el heap y mantener variables que los refieran, eliminando la sobrecarga de mantener múltiples copias de datos extensos.

Tipos Predefinidos

La librería estándar de ..NET ofrece un conjunto amplio de tipos de datos. Las siguientes tablas muestran los tipos de datos básicos más comunes.

Tabla 1.7. Tipos de datos integrales

Tipo	Rango	Tamaño
sbyte	-128 al 127	Entero de 8 bits con signo
byte	0 al 255	Entero de 8 bits sin signo
char	U+0000 al U+ffff	Carácter UNICODE de 16 bits
short	-32,768 al 32,767	Entero de 16 bits con signo
ushort	0 al 65,535	Entero de 16 bits sin signo
int	-2,147,483,648 a 2,147,483,647	Entero de 32 bits con signo
uint	0 al 4,294,967,295	Entero de 32 bits sin signo
long	-9,223,372,036,854,775,808 al 9,223,372,036,854,775,807	Entero de 64 bits con signo
ulong	0 al 18,446,744,073,709,551,615	Entero de 64 bits sin signo

Tabla 1.8. Tipos de datos de punto flotante

Tipo	Rango aproximado	Precisión	Tamaño
float	±1.5 × 10-45 al ±3.4 × 1038	7 dígitos	32 bits
double	±5.0 × 10-324 al ±1.7 × 10308	15 a16 dígitos	64 bits

Tabla 1.9. Tipos de datos decimales

Tipo	Rango aproximado	Precisión	Tamaño
decimal	1.0 × 10-28al 7.9 × 1028	28 a29 dígitos significativos	128 bits

Tabla 1.10. Tipo booleano

Tipo	Valores
bool	true y false

Tabla 1.11. Tipo carácter

Tipo	Rango	Tamaño
char	U+0000 al U+ffff	Carácter Unicode de 16 bits

Tabla 1.12. Tipos referencia

Tipo	Descripción
object	La clase raíz de la que todos los tipos de datos, predefinidos y nopredefinidos, tipo valor o tipo referencia, derivan.
string	Cadena de carácteres Unicode.

Dado que el tipo string es un tipo referencia, podríamos esperar que sea sencillo cometer errores de programación al asignar una variable string a otra, y luego al modificar la cadena de la primera estaríamos modificando, quizá inadvertidamente la segunda. Sin embargo, el tipo string presenta una característica particular que evita este tipo de error. Cuando se realiza cualquier operación que modifica el contenido de una variable string, se crea un nuevo objeto de datos, manteniendo el valor de la variable original sin cambios. Luego, la regla es, una vez creado un objeto string, el valor que almacena no puede ser modificado. El siguiente código ejemplifica esta característica.

```
using System;
class MainClass {
      public static void Main(string[] args) {
            string s1 = "Una cadena";
            string s2 = s1;
            Console.WriteLine("s1 es " + s1);
            Console.WriteLine("s2 es " + s2);
            s1 = "Otra cadena";
            Console.WriteLine("s1 es ahora " + s1);
            Console.WriteLine("s2 es ahora " + s2);
      }
}
```

Definidos por el Programador

Todos los tipos de datos se clasifican en dos grupos:

- Tipo valor: Simples (incluyendo los enumerados) y las estructuras.
- Tipo referencia: Clases, interfaces y delegados.

La declaración y uso de las estructuras, enumerados y clases es muy similar a C++. Los siguientes formatos corresponden a la declaración de éstos:

```
[modificadores] struct <nombre del tipo> { <cuerpo de la estructura> }
[modificadores] enum <nombre del tipo> [:<tipo entero>] { <declaración de las constantes>}
[modificadores] class <nombre del tipo> [: <clases e interfaces>] { <cuerpo de la clase> }
```

El siguiente código muestra un ejemplo simple del uso de estos tres tipos:

```
using System;

enum Sexo {
       Masculino,
       Femenino
}

struct Persona {
       public string nombre;
       public int edad;
       public Sexo sexo;
}

class MainClass {
       public static void Main(string[] args) {
              Persona p = new Persona();
              p.nombre = "Jose";
              p.edad = 25;
              p.sexo = Sexo.Masculino;
              string saludo = CrearSaludo(p);
              Console.WriteLine(saludo);
       }

       public static string CrearSaludo(Persona p) {
              string saludo = "Bueno dias";
              switch(p.sexo) {
                     case Sexo.Masculino:
                            saludo += " Sr. ";
                            break;
                     case Sexo.Femenino:
                            saludo += " Sra. ";
                            break;
              }
              saludo += p.nombre;
              return saludo;
       }
}
```

Nótese que los datos miembros de una estructura se deben declarar public para ser accedidos directamente. Tanto para las clases como para las estructuras cuando uno de sus miembros no especifica el nivel de acceso, mediante un modificador, se asume por defecto *private*. C++ asume por defecto private para las clases y public para las estructuras. Aunque para la inicialización de una estructura se utiliza el operador new, es importante recordar que los datos de ésta se encuentran en el stack, no en el heap. Existen otras diferencias entre las estructuras de C++ y C# que van más allá de una introducción al lenguaje.

Nótese que el formato para acceder a los elementos de un enumerado difiere del de C++. C# no permite definir un tipo enumerado anónimo, como sí lo permite C++. A diferencia de C++, el nombre de un tipo enumerado define un espacio de nombres. Es por ello que para acceder a un elemento del enumerado se requiere el formato:

```
<nombre del enumerado>.<nombre de la constante>
```

Las interfaces y los delegados son temas avanzados que van más allá de una introducción al lenguaje.

Conversión entre Tipos

El siguiente código produce un error poco claro:

```
byte valor1 = 50;
byte valor2 = 100;
byte total;
total = valor1 + valor2;
Console.WriteLine(total);
```

El error es: CS0029: Cannot implicitly convert type 'int' to 'byte'. Esto se debe que, debido a que la suma de dos números tipo byte, cuyo rango de valores va del cero al 255, puede producir fácilmente un número mayor a 255, lo que requeriría un tipo de dato entero de por lo menos dos bytes. Debido a esto, la suma de tipos byte en C# retorna un valor entero tipo int, lo que origina el error en el código, dado que el resultado de la suma, un int, se intenta asignar a una variable byte, lo que podría producir una pérdida de información. Luego, para solucionar este problema requerimos de una conversión de un tipo de dato a otro.

En C# existen dos formas de conversión del valor de una variable: implícita y explícita. La conversión implícita es realizada en forma automática por el compilador del lenguaje, sin que el programador lo solicite, solamente en los casos donde es seguro que no se perderá información o se modificará el valor original. En los demás casos, se requerirá una conversión explícita, donde el programador solicita la conversión mediante una sintaxis especial.

El siguiente es un ejemplo de conversión implícita:

```
byte valor1 = 50;
byte valor2 = 100;
long total;
total = valor1 + valor2;
Console.WriteLine(total);
```

Dado que toda suma de dos valores tipo byte siempre pueden ser almacenados en una variable tipo long, el compilador realiza una conversión implícita. La tabla 1.13 muestra las conversiones implícitas para los tipos de datos primitivos.

Tabla 1.13. Conversiones implícitas para tipos de datos primitivos

Desde	A
sbyte	short, int, long, float, double, o decimal
byte	short, ushort, int, uint, long, ulong, float, double, o decimal
short	int, long, float, double, o decimal
ushort	int, uint, long, ulong, float, double, o decimal
int	long, float, double, o decimal
uint	long, ulong, float, double, o decimal
long	float, double, o decimal
char	ushort, int, uint, long, ulong, float, double, o decimal
float	Double
ulong	float, double, o decimal

Para realizar una conversión explícita requerimos realizar una operación cast, al igual que en C++. El siguiente código realiza una conversión explícita:

```
long valor1 = 30000;
int valor2 = (int)valor1;
```

Es importante recordar que el riesgo de pérdida o modificación de valores en este tipo de operación corre por cuenta del programador.

Para los datos primitivos, sólo se permiten las conversiones entre tipos enteros y carácteres. No es posible realizar una conversión ni implícita ni explícita desde estos tipos a un booleano y viceversa.

Para convertir un tipo primitivo a una cadena se puede utilizar el método ToString. Este método es heredado por todos los tipos de datos de la clase base object. El siguiente código muestra la conversión de un entero a una cadena.

```
int i = 10;
string s = i.ToString();
```

El método ToString es llamado automáticamente cuando se concatena, con una operación de suma '+', cualquier objeto de datos, variables o constantes (la declaración de éstas se verán más adelante), con un objeto tipo string. El siguiente ejemplo muestra este caso.

```
int entero = 100;
double real = 534.65;
bool booleano = true;
string cadena = "entero=" + entero + ", real=" + real +
", booleano=" + booleano;
Console.WriteLine("cadena=" + cadena);
```

Al ejecutar este código se mostrará en la ventana de consola lo siguiente:

```
cadena=entero=100, real=534.65, booleano=True
```

Para convertir una cadena a un tipo primitivo es posible utilizar el método estático Parse de estos. El siguiente código convierte un objeto string a un objeto int.

```
string s = "123";
int e = int.Parse(s);
```

Este código refuerza la idea de que "todos" los tipos de datos en C# son objetos, aun los datos primitivos e inclusive las constantes literales, por lo que el siguiente código sería válido.

```
string s = 10.ToString();
```

Sin embargo, lo que ocurre en el fondo en este código es la creación de un objeto temporal en el heap que encajone un valor entero, de forma que pueda llamarse al método requerido. A este proceso se llama encajonamiento o boxing. También es posible realizar un boxing explícitamente, como en el siguiente código.

```
int i = 123;
object obj = i;
Console.WriteLine("obj = " + obj);
```

Es interesante notar que al ejecutarse este código se imprime en pantalla la siguiente línea.

```
obj = 123
```

La variable tipo referencia o realmente referencia a un objeto tipo int pero en heap. Dado que el método ToString originario de la clase object es sobreescrito por su clase derivada int, su llamada se realiza polimórficamente, lo que origina el resultado mostrado. Este proceso se muestra en la figura 1.1.

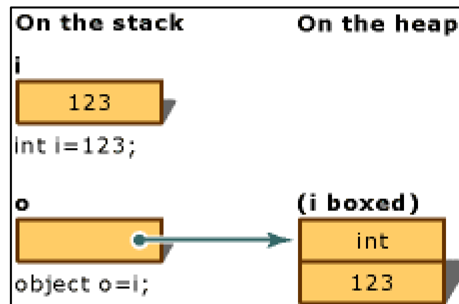

Figura 1.1. Proceso de Boxing

Un proceso boxing puede realizarse sobre cualquier objeto tipo valor, como los enteros y las estructuras. El proceso inverso se llama unboxing. El siguiente código muestra un ejemplo de su uso.

```
int i = 20;
object obj = i;
int j = (int)obj;
j++;
Console.WriteLine("i=" + i + ", j=" + j);
```

La salida producida al ejecutar este código es la siguiente.

```
i=20, j=21
```

Como se puede ver, el modificar el valor de la variable j no afecta a la variable original i. Igualmente, el objeto entero encajonado por la variable obj es una copia del valor de la variable i, esto es, obj no es una referencia a la variable i original, obj únicamente contiene una copia del valor original de i.

Si bien es poco usual que se requiera realizar un proceso de boxing o unboxing explícitamente, el que existan los mecanismos que permitan hacerlo permitiría, en caso de requerirlo un programador, trabajar con cualquier tipo de dato, inclusive los primitivos, de manera uniforme.