

Notas de Programación Orientada a Objetos con C++ Java Ruby Python C# D Scala

POO Handbook 2025

Carlos Alberto Fernández y Fernández

*Instituto de Computación
Universidad Tecnológica de la Mixteca*

*“Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.”*

*“Considerad vuestra estirpe:
hechos no fuisteis para vivir como brutos,
sino para perseguir virtud y conocimiento.”*

– DANTE ALIGHIERI, LA DIVINA COMEDIA

Universidad Tecnológica de la Mixteca
Notas de POO 2025
?? páginas

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Índice general

Índice de cuadros

Índice de figuras

Índice de código fuente

Parte I

Introducción

Capítulo 1

Lenguajes y Ambientes sugeridos para desarrollo

1.1 Lenguajes de programación

Existe una infinidad de lenguajes de programación. Además de los cubiertos de manera general en este material podemos mencionar algunos como:

- **Groovy**. Es un lenguaje orientado a objetos y dinámico, similar a Python, Ruby, Perl y Smalltalk pero que es dinámicamente compilado hacia bytecodes de la máquina virtual de Java.



```
1 println "Hola Mundo"
```

- **JRuby**. Es una implementación en Java del intérprete de Ruby. Su alta integración con Java permite completo acceso en los dos sentidos entre código Java y Ruby.

```
1 puts "Hola Mundo"
```

- **Jython/JPython**. Una implementación de Python en Java. Programas en Jython pueden importar y usar clases en Java.



```
1 print("Hola Mundo")
```

- **Kotlin.** Un lenguaje orientado a objetos para JVM, para aplicaciones del lado del servidor, Android y compilación a JavaScript.

```
1 fun main() {  
2     println("Hola Mundo")  
3 }
```

- **IronPython.** Es una implementación de Python en .NET y Mono. Permite el uso de bibliotecas de .NET y una fácil interoperabilidad con lenguajes como C#.

```
1 print("Hola Mundo desde .NET")
```

- **Dart.** Es un lenguaje de programación de código abierto, desarrollado por Google, optimizado para la creación de aplicaciones web, móviles (con Flutter), de escritorio y del lado del servidor.

```
1 void main() {  
2     print('Hola Mundo');  
3 }
```

- **Go.** Conocido también como Golang, es un lenguaje compilado y concurrente, desarrollado por Google. Es apreciado por su simplicidad y eficiencia en sistemas distribuidos.

```
1 package main  
2 import "fmt"  
3 func main() {
```

```
4     fmt.Println("Hola Mundo")
5 }
```

- **Swift.** Creado por Apple, es un lenguaje multiparadigma para el desarrollo de aplicaciones en iOS, macOS, watchOS y tvOS. Se enfoca en seguridad y rendimiento.

```
1 print("Hola Mundo")
```

- **Cobra.** Lenguaje inspirado en Python y Ruby, pero con tipos estáticos, generación de código para .NET/Mono, ligado dinámico, contratos y soporte de pruebas de unidad.

```
1 class Hola
2     def main
3         print "Hola Mundo"
```

- **Fantom.** Es un lenguaje portable y orientado a objetos, diseñado para ejecutarse en múltiples plataformas (JVM, .NET y JavaScript). Más información en: <http://fantom.org/>

```
1 class Hola {
2     static Void main() {
3         echo("Hola Mundo")
4     }
5 }
```

- **Elixir.** Es un lenguaje funcional, concurrente y distribuido que se ejecuta en la máquina virtual de Erlang (BEAM). Es usado en sistemas tolerantes a fallos y aplicaciones web escalables.

```
1 IO.puts "Hola Mundo"
```

En la tabla 1.1 podemos ver una tabla con algunas características de los lenguajes antes mencionados.

Cuadro 1.1. Comparación de lenguajes de programación

Lenguaje	Paradigma principal	Plataforma base	Año de creación
Groovy	Orientado a objetos, dinámico	JVM (Java Virtual Machine)	2003
JRuby	Orientado a objetos (Ruby)	JVM	2001
Jython	Imperativo, orientado a objetos (Python)	JVM	1997
Kotlin	Orientado a objetos y funcional	JVM, Android, compilación a JS	2011
IronPython	Imperativo, orientado a objetos (Python)	.NET, Mono	2006
Dart	Orientado a objetos	Web, Flutter (móvil), escritorio	2011
Go (Golang)	Imperativo, concurrente, orientado a objetos ligero	Nativo (compilado)	2009
Swift	Multiparadigma (OO, funcional)	iOS, macOS, watchOS, tvOS	2014
Cobra	Orientado a objetos con contratos	.NET, Mono	2006
Fantom	Orientado a objetos, multiplataforma	JVM, .NET, JavaScript	2005
Elixir	Funcional, concurrente, distribuido	BEAM (Erlang VM)	2011

1.1.1 IDEs

Los *Entornos de Desarrollo Integrados* (IDE, por sus siglas en inglés) son herramientas que reúnen en una sola aplicación diversos componentes necesarios para programar: editores de código, compiladores, depuradores, asistentes gráficos y gestores de proyectos. Su objetivo es aumentar la productividad del desarrollador y facilitar el mantenimiento de proyectos grandes.

Eclipse

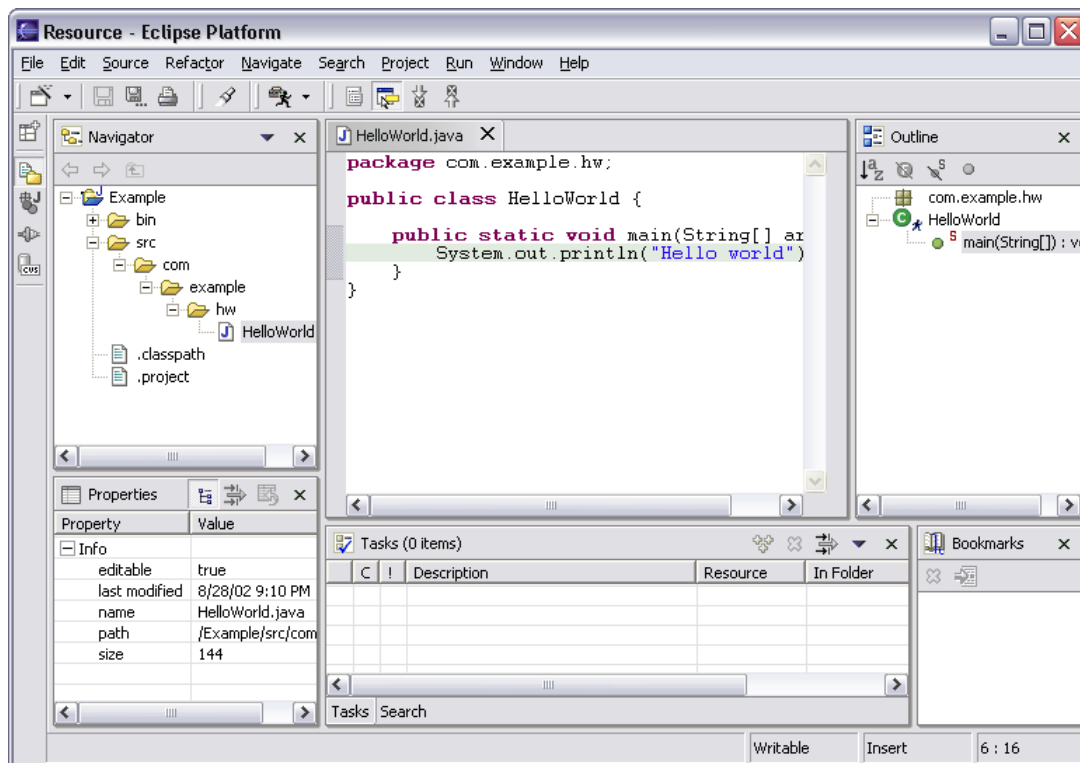
Eclipse es desarrollado como un proyecto de código abierto lanzado en noviembre de 2001 por IBM, Object Technology International y otras compañías. El objetivo era desarrollar una plataforma abierta de desarrollo. Fue planeada para ser extendida mediante plug-ins.



Es desarrollada en Java, por lo que puede ejecutarse en un amplio rango de sistemas operativos. También incorpora facilidades para desarrollar en Java, aunque es posible instalarle plug-ins para otros lenguajes como C/C++, PHP, Ruby, Haskell, etc. Incluso antiguos lenguajes como Cobol tienen extensiones disponibles para Eclipse [1]:

- Eclipse + JDT = Java IDE
- Eclipse + CDT = C/C++ IDE
- Eclipse + PHP = PHP IDE
- Eclipse + JDT + CDT + PHP = Java, C/C++, PHP IDE

Trabaja bajo “*workbenches*” que determinan la interfaz del usuario centrada alrededor del editor, vistas y perspectivas.



Los recursos son almacenados en el espacio de trabajo (workspace) el cual es un folder almacenado normalmente en el directorio de Eclipse. Es posible manejar diferentes áreas de trabajo.

Eclipse, sus componentes y documentación pueden ser obtenidos de: www.eclipse.org

Mono

Mono es una alternativa de software libre patrocinada por Novell. Implementa principalmente un compilador para C# y el *Common Language Runtime* de .NET. Incluye un IDE y existen versiones para plataformas distintas a Windows. Su IDE es **MonoDevelop**, el cual facilita la creación de aplicaciones multiplataforma.

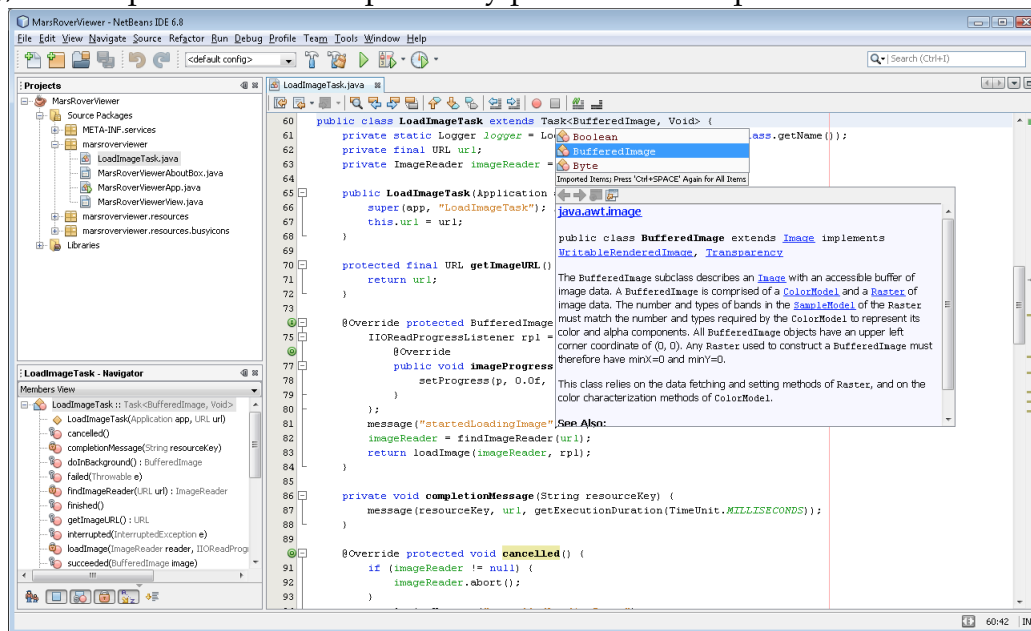


NetBeans

NetBeans es una plataforma de desarrollo y un IDE multilenguaje. Originalmente creado para desarrollo en Java y adquirido por Sun Microsystems. El IDE es actualmente de código abierto y disponible en www.netbeans.org.



Creado en 1996 como un proyecto universitario en la Universidad Karlova (Praga), fue adoptado más tarde por Sun y posteriormente por Oracle.

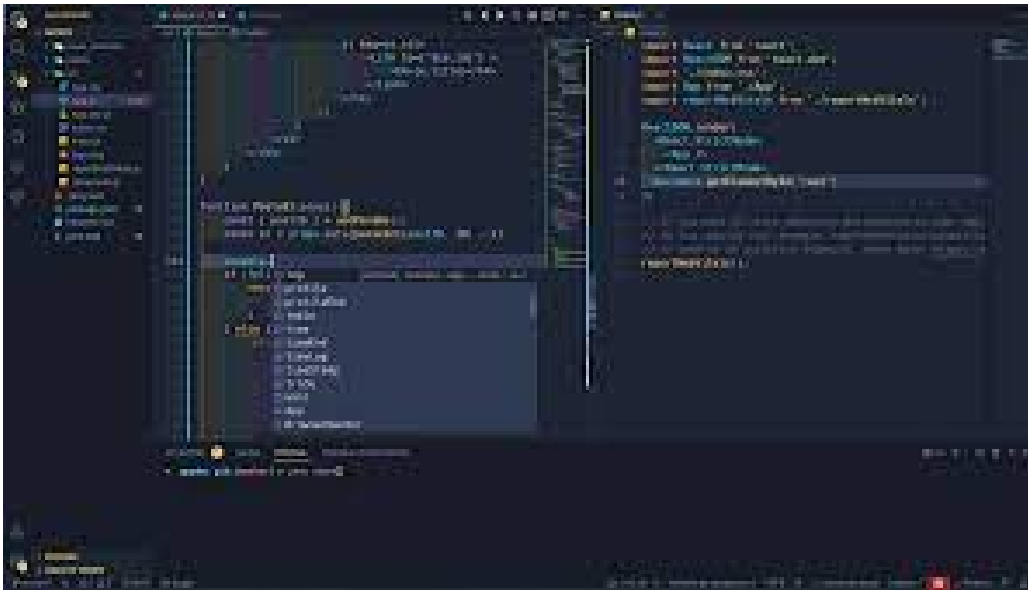


Soporta una variedad de lenguajes y tecnologías como C/C++, Java (SE, EE, ME), Ruby y PHP. NetBeans destaca por integrar de manera nativa un diseñador visual de interfaces gráficas (GUI Builder), útil en el desarrollo rápido de aplicaciones de escritorio.

Visual Studio Code

Visual Studio Code es un editor gratuito y de código abierto desarrollado por Microsoft.¹

¹<https://code.visualstudio.com/>



Se caracteriza por ser ligero, extensible y multiplataforma (Windows, Linux, macOS). Algunas características principales son:

- Interfaz de usuario intuitiva y personalizable.
- Soporte integrado para depuración paso a paso.
- Amplo ecosistema de extensiones para lenguajes, depuradores y control de versiones.
- Integración nativa con Git y GitHub.
- Autocompletado inteligente mediante *IntelliSense*.

Hoy en día, Visual Studio Code es probablemente el editor más popular en la comunidad de desarrolladores.

Editores asistidos por IA

La inteligencia artificial ha revolucionado la forma en que escribimos código. Nuevas herramientas integran modelos de lenguaje avanzados directamente en el flujo de trabajo del desarrollador.

- **Cursor.** Es un editor de código construido sobre VS Code que integra IA de manera nativa. Permite generar código, editar bloques existentes y chatear con el proyecto entero para entender el contexto. Al ser un fork de VS Code, mantiene compatibilidad con sus extensiones.

- **Antigravity.** Es un asistente de codificación agéntico avanzado desarrollado por el equipo de Google Deepmind. Diseñado para trabajar como un par programador, puede realizar tareas complejas, modificar múltiples archivos y ejecutar comandos, ofreciendo un nivel de autonomía superior en la resolución de problemas de programación. Al igual que Cursor, es un fork de VS Code, por lo que aprovecha la compatibilidad con sus extensiones con los componentes de este.

Algunos editores ligeros

Si no se desea usar IDEs completos como Eclipse o NetBeans, existen editores ligeros que ofrecen rapidez y flexibilidad:

- Geany.²
- Sublime Text.³
- Atom.⁴
- Brackets.⁵

Estos editores no incluyen compiladores, por lo que dependen de las herramientas de línea de comandos instaladas en el sistema.

Otros ejemplos de frameworks

Algunos de los principales frameworks usados para desarrollo Web son:

- **Ruby on Rails.** Framework gratuito para desarrollo de aplicaciones Web en



Ruby.

- **Merb.** Framework para desarrollo web en Ruby, diseñado con enfoque en concurrencia y modularidad.

²<https://www.geany.org/>

³<https://www.sublimetext.com/>

⁴<https://atom.io/>

⁵<http://brackets.io/>

- **Django.** Framework open source para desarrollo de aplicaciones web con



Python.

- **Grails.** Framework open source para el lenguaje Groovy, basado en la JVM.



- **SproutCore.** Framework open source para aplicaciones web con JavaScript, orientado a experiencias similares a aplicaciones de escritorio. Fue utilizado por Apple en proyectos como MobileMe.
- **Lift.** Framework para desarrollo web en Scala. Aprovecha la JVM y la biblioteca de Java, garantizando compatibilidad y escalabilidad.

Comparación de IDEs

En la tabla 1.2, se presenta una breve comparación de los IDEs.

Cuadro 1.2. Breve comparación de IDEs

IDE	Características principales	Lenguajes base	Licencia
Eclipse	Altamente extensible mediante plugins; multiplataforma; soporte a múltiples lenguajes	Java, C/C++, PHP, Ruby, entre otros	EPL (open source)
MonoDevelop	Orientado a C# y .NET multiplataforma; integración con GTK#	C#, F#	MIT/X11
NetBeans	IDE completo con GUI Builder; soporte para múltiples lenguajes; enfoque educativo y empresarial	Java, C/C++, PHP, Ruby	Apache License 2.0
VS Code	Editor ligero; extensible; integración con Git; depuración integrada	Multilenguaje (a través de extensiones)	MIT
Editores ligeros	Simples, rápidos, personalizables; requieren compiladores externos	Multilenguaje (sin integración nativa)	Variada

1.2 Lenguajes estáticos y dinámicos

Los lenguajes de programación pueden clasificarse, entre otros criterios, según su tipado en dos categorías principales: estáticos y dinámicos. La diferencia central radica en el momento en que se verifican los tipos de datos y la validez de las operaciones.

En los **lenguajes estáticos**, las variables y sus tipos se comprueban en tiempo de compilación. Esto permite detectar errores de tipo y de sintaxis antes de la ejecución, lo que conduce a programas más seguros y predecibles. Además, la vinculación de variables y funciones ocurre en esta fase, lo que suele mejorar la eficiencia del código resultante. Ejemplos comunes son *C*, *C++*, *Java*, *Go* y *C#*.

En contraste, los **lenguajes dinámicos** realizan la verificación en tiempo de ejecución. Los errores pueden aparecer solo cuando el programa se ejecuta, lo que implica mayor riesgo de fallos inesperados. No obstante, esta flexibilidad facilita cambios rápidos en el código y acelera el desarrollo. Lenguajes representativos son *Python*, *JavaScript* y *Ruby*.

En síntesis, los lenguajes estáticos tienden a ser preferidos en proyectos grandes y críticos, donde la robustez es esencial, mientras que los dinámicos son útiles en entornos donde prima la agilidad y la experimentación.

1.3 Documentación automática de código

La documentación automática de código es un proceso esencial en el desarrollo de software, donde se generan automáticamente descripciones y explicaciones del código fuente sin intervención manual por parte del programador.

La documentación automática es llevada a cabo por herramientas especializadas, las cuales analizan el código fuente y extraen información relevante sobre clases, métodos, variables y sus interacciones. Estos detalles se utilizan para generar documentos legibles y coherentes que proporcionan una guía comprensiva sobre la estructura y funcionamiento del programa.

Al utilizar la documentación automática, se logra mejorar la comprensión del código tanto para los propios desarrolladores como para otros colaboradores o usuarios potenciales del software. Esto es especialmente valioso en proyectos de gran envergadura, donde el código puede volverse complejo y difícil de entender con facilidad.

Además de facilitar la comprensión, la documentación automática también promueve una mayor eficiencia en el proceso de desarrollo y mantenimiento del software. Al tener documentación actualizada y generada automáticamente, los desarrolladores pueden ahorrar tiempo en la creación y actualización manual de la documentación.

Es importante resaltar que, si bien la documentación automática es una herramienta poderosa, no reemplaza por completo la documentación manual y las buenas prácticas de programación. Es fundamental que los programadores también incluyan comentarios significativos en el código para explicar decisiones de diseño, funcionalidades importantes y otras consideraciones relevantes que no pueden ser extraídas automáticamente.

La documentación automática de código es un proceso clave en el desarrollo

de software, que utiliza herramientas especializadas para generar descripciones detalladas del código fuente sin intervención manual. Esta práctica mejora la comprensión del código, fomenta la eficiencia en el desarrollo y es especialmente útil en proyectos complejos. Sin embargo, se recomienda complementarla con comentarios manuales para brindar una documentación más completa y detallada.

Hay una variedad de herramientas disponibles para la documentación automática de código, y cada una tiene sus propias características y ventajas. Algunas de las herramientas más populares incluyen:

- Doxygen: una herramienta de código abierto que genera documentación HTML, XML, LaTeX y RTF a partir de comentarios en el código fuente.
- Javadoc: una herramienta de código abierto que genera documentación HTML a partir de comentarios en el código fuente de Java.
- Sphinx: una herramienta de código abierto que genera documentación HTML, LaTeX y RST a partir de comentarios en el código fuente.

1.3.1 Documentación automática de código en Python

Python, al ser un lenguaje altamente utilizado y apreciado por su simplicidad y legibilidad, fomenta el uso de la documentación para facilitar la comprensión y el mantenimiento de proyectos de software. En este contexto, la documentación automática en Python se realiza mediante el uso de herramientas específicas, como *Pydoc*, *Sphinx*, o *Doxygen*, que analizan el código fuente y extraen información relevante para generar documentación coherente.

Pydoc es una herramienta incorporada en Python que permite generar documentación automática directamente desde el código fuente y los comentarios del programa.

Pydoc analiza el código Python y busca *docstrings*, que son cadenas de texto colocadas como primeras líneas de definiciones de módulos, clases, funciones y métodos. Estas *docstrings* contienen información detallada sobre el propósito, la funcionalidad y el uso de cada componente del código. La herramienta *Pydoc* extrae estas *docstrings* y las convierte en una documentación estructurada y legible.

A continuación, se presenta un ejemplo de código documentado utilizando *docstrings* y cómo *Pydoc* lo utiliza para generar documentación automática:

```
1  # Ejemplo de función para calcular el área de un círculo
2  def calcular_area_circulo(radio):
3      """
4      Calcula el área de un círculo dado su radio.
5
6      Parámetros:
```

```
7         radio (float): El radio del círculo.
8
9     Retorna:
10         float: El área del círculo.
11     """
12     area = 3.1416 * radio ** 2
13     return area
14
```

Listing 1. Ejemplo de documentación en Python.

En este ejemplo, la función calcular *area_circulo* está documentada utilizando una *docstring*. La *docstring* describe claramente el propósito de la función, los parámetros que acepta y el tipo de valor que deben tener, así como el valor de retorno que se puede esperar. Estas descripciones son esenciales para que *Pydoc* pueda generar una documentación detallada.

Para generar la documentación utilizando *Pydoc*, se debe ejecutar el siguiente comando en la línea de comandos:

```
python -m pydoc nombre_del_archivo.py
```

o directamente *pydoc* :

```
pydoc -w nombre_del_archivo.py
```

Para ver la documentación podemos dar el nombre del módulo o archivo:

```
pydoc nombre.py
```

Inclusive ver localmente toda la documentación que tengamos disponible en nuestra localmente en nuestra computadora, lanzando un servidor temporal:

```
pydoc -p 8001
```

Pydoc es una herramienta útil para generar documentación automática básica para el código Python. Sin embargo, es importante tener en cuenta que no es una herramienta de documentación completa.

Los compiladores de C++ son normalmente compatibles con C, y se puede usar el de preferencia, pero si usan la terminal de linux se compila:

```
g++ -ofoo.elfoo.cpp
```

o, desde el compilador de C:

```
gcc -ofoo.elfoo.cpp -lstdc++
```


Capítulo 2

Introducción a características de C++

Ahora comentaremos algunas características de C++ que no tienen que ver directamente con la programación orientada a objetos.

2.1 Comentarios en C++

Los comentarios en C son:

```
/* comentario en C */
```

En C++ los comentarios pueden ser además de una sola línea:

```
// este es un comentario en C++
```

Acabando el comentario al final de la línea, lo que quiere decir que el programador no se preocupa por cerrar el comentario.

2.2 Flujo de entrada/salida

En C, la salida y entrada estándar estaba dada por *printf* y *scanf* principalmente (o funciones similares) para el manejo de los tipos de datos simples y las cadenas. En C++ se proporcionan a través de la biblioteca *iostream*, la cual debe ser insertada a través de un *#include*. Las instrucciones son:

- *cout*. Utiliza el flujo de salida estándar. Que se apoya del operador `<`, el cual se conoce como operador de inserción de flujo *colocar en*.
- *cin*. Utiliza el flujo de entrada estándar. Que se apoya del operador `>`, conocido como operador de extracción de flujo *obtener de*.

Los operadores de inserción y de extracción de flujo no requieren cadenas de formato (%s,%f), ni especificadores de tipo. C++ reconoce de manera automática que tipos de datos son extraídos o introducidos.

En el caso del operador de extracción `>>` no se requiere el operador de dirección `&`.

De tal forma un código de desplegado con *printf* y *scanf* de la forma (usando el área de nombres estándar):

```
printf("Número: ");
```

```
scanf("%d", &num);
```

```
printf("El valor leído es: " "%d\n", num);
```

Sería en C++ de la siguiente manera:

```
cout << "Número";
```

```
cin >> num;
```

```
cout << "El valor leído es: " << num << '\n';
```

2.3 Funciones en línea

Las funciones en línea, se refiere a introducir un calificador *inline* a una función de manera que le sugiera al compilador que genere una copia del código de la función en lugar de la llamada.

Ayuda a reducir el número de llamadas a funciones reduciendo el tiempo de ejecución en algunos casos, pero en contraparte puede aumentar el tamaño del programa.

A diferencia de las macros, las funciones *inline* si incluyen verificación de tipos y son reconocidas por el depurador. Un depurador puede ayudar a encontrar un error de lógica resultado de usar una macro, pero no puede atribuir los errores a alguna macro.

Las funciones *inline* deben usarse sólo para funciones chicas que se usen frecuentemente.

El compilador desecha las solicitudes *inline* para programas que incluyan un ciclo, un *switch* o un *goto*. Tampoco se consideran si no tienen *return* (aunque no regresen valores) o si contienen variables de tipo *static*. Además, lógicamente no genera una función *inline* para funciones recursivas.

Sintaxis	
1	<code>inline <declaración de la función></code> -

Ejemplo:

```

1 inline float suma (float a, float b) {
2     return a+b;
3 }
4
5 inline int max( int a, int b) {
6     return (a > b) ? a : b;
7 }

```

Nota: Las funciones *inline* tienen conflictos con los prototipos, así que deben declararse completas sin prototipo en el archivo .h. Además, si la función en línea hace uso de otra función, en donde se expanda la función en línea debe tener los *include* correspondientes a esas funciones utilizadas.

2.4 Declaraciones de variables

Mientras que en C, las declaraciones deben ir en la función antes de cualquier línea ejecutable, en C++ pueden ser introducidas en cualquier punto, con la condición lógica de que la declaración esté antes de la utilización de lo declarado.

En algunos compiladores podía declararse una variable en la sección de inicialización de la instrucción *for*, pero es incorrecto declarar una variable en la expresión condicional del *while*, *do-while*, *for*, *if* o *switch*. El actual estándar de C++ no permite la declaración de variables dentro del *for*.

Ejemplo:

O usando el área de nombres estándar:

El alcance de las variables en C++ es por bloques. Una variable es vista a partir de su declaración y hasta la llave “}” del nivel en que se declaró. Lo cual quiere decir que las instrucciones anteriores a su declaración no pueden hacer uso de la variable, ni después de finalizado el bloque.

```
1  #include <iostream>
2
3  int main() {
4      int i=0;
5
6      for (i=1; i<10; i++){
7          int j=10;
8          std::cout<<i<<" j: "<<j<<std::endl;
9      }
10
11     std::cout<<"\n al salir del ciclo: "<<i;
12
13     return 0;
14 }
```

Listing 2. Ejemplo declaración de variables

2.5 Operador de resolución de alcance

Se puede utilizar el operador de resolución de alcance `::` se refiere a una variable (variable, función, tipo, enumerador u objeto), con un alcance de archivo (variable global). No tiene alcance de bloque, aunque esten variables definidas con el mismo nombre en varios niveles de bloques.

Esto le permite al identificador ser visible aún si el identificador se encuentra oculto.

Ejemplo:

```
1  float h;
2
3  void g(int h) {
4      float a;
5      int b;
6
7      a>::h;           // a se inicializa con la variable global h
8
9      b=h;             // b se inicializa con la variable local h
10 }
```

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i=0;
7
8      for (i=1; i<10; i++){
9          int j=10;
10         cout<<i<<" j: "<<j<<endl;
11     }
12
13     cout<<"\n al salir del ciclo: "<<i;
14
15     return 0;
16 }
```

Listing 3. Ejemplo declaración de variables usando el área de nombres estándar

2.6 Valores por default

Las funciones en C++ pueden tener valores por default. Estos valores son los que toman los parámetros en caso de que en una llamada a la función no se encuentren especificados.

Los valores por omisión deben encontrarse en los parámetros que estén más a la derecha. Del mismo modo, en la llamada se deben empezar a omitir los valores de la extrema derecha.

Ejemplo:

C++ no permite la llamada omitiendo un valor antes de la extrema derecha de los argumentos:

```
punto( , 8);
```

Otro ejemplo de valores o argumentos por default:

```
1
2  #include <iostream>
3
4  using namespace std;
```

```
1  #include <iostream>
2
3  using namespace std;
4
5  int punto(int=5, int=4);
6
7  int main () {
8
9      cout<<"valor 1: "<<punto()<<'\\n';
10     cout<<"valor 2: "<<punto(1)<<'\\n';
11     cout<<"valor 3: "<<punto(1,3)<<'\\n';
12     return 0;
13 }
14
15 int punto( int x, int y){
16
17     if (y!=4)
18         return y;
19     if (x!=5)
20         return x;
21     return x+y;
22 }
```

Listing 4. Ejemplo de valores por default

```
5
6  int b=1;
7  int f(int);
8  int h(int x=f(b));           // argumento default f(:,b)
9
10 int main () {
11     b=5;
12     cout<<b<<endl;
13     {
14         int b=3;
15         cout<<b<<endl;
16         cout<<h();           //h(f(:,b))
17     }
18
19     return 0;
```

```
20 }
21
22 int h(int z) {
23     cout<<"Valor recibido: "<<z<<endl;
24     return z*z;
25 }
26 int f(int y) {
27     return y;
28 }
```

Listing 5. Valores o argumentos por default.

2.7 Parámetros por referencia

En C todos los pasos de parámetros son por valor, aunque se pueden enviar parámetros "por referencia" al enviar por valor la dirección de un dato (variable, estructura, objeto), de manera que se pueda acceder directamente el área de memoria del dato del que se recibió su dirección. Aunque se tienen que manejar en algunas ocasiones como apuntadores.

C++ introduce parámetros por referencia **reales**. La manera en que se definen es agregando el símbolo & de la misma manera que se coloca el *: después del tipo de dato en el prototipo y en la declaración de la función.

Ejemplo:

```
1 // Comparando parámetros por valor, por valor con apuntadores ("referencia"),
2 // y paso por referencia real
3
4 #include <iostream>
5 using namespace std;
6
7 int porValor(int);
8 void porApuntador(int *);
9 void porReferencia( int &);
10
11 int main() {
12     int x=2;
13     cout << "x= " << x << " antes de llamada a porValor \n"
14         << "Regresado por la función: " << porValor(x) << endl
15         << "x= " << x << " despues de la llamada a porValor\n\n";
16
17     int y=3;
```

```

18     cout << "y= " << y << " antes de llamada a porApuntador\n";
19     porApuntador(&y);
20     cout << "y= " << y << " despues de la llamada a porApuntador\n\n";
21
22     int z=4;
23     cout << "z= " << z << " antes de llamada a porReferencia \n";
24     porReferencia(z);
25     cout<< "z= " << z << " despues de la llamada a porReferencia\n\n";
26     return 0;
27 }
28
29 int porValor(int valor){
30     return valor*=valor;           //parámetro no modificado
31 }
32
33 void porApuntador(int *p){
34     *p *= *p;                       // parámetro modificado
35 }
36
37 void porReferencia( int &r){
38     r *= r;                         //parámetro modificado
39 }
40

```

Listing 6. Otro ejemplo Parámetros por referencia.

Notar que no hay diferencia en el manejo de un parámetro por referencia y uno por valor, lo que puede ocasionar ciertos errores de programación.

Variables de referencia

También puede declararse una variable por referencia que puede ser utilizada como un seudónimo o alias. Ejemplo:

```

1  int max=1000, &sMax=max;           //declaro max y sMax es un alias de max
2  sMax++;                           //incremento en uno max a través de su alias

```

Esta declaración no reserva espacio para el dato, pues es como un apuntador pero se maneja como una variable normal. No se permite reasignarle a la variable por referencia otra variable. La variable por referencia debe ser inicializada en el momento de su declaración.

Ejemplo:

```

1  // variable por referencia
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int x=2, &y=x, z=8;
8
9      cout << "x= " << x << endl
10         << "y= " << y << endl;
11
12     y=10;
13     cout << "x= " << x << endl
14         << "y= " << y << endl;
15     // Reasignar no esta permitido
16     //      \& y= &z;
17     //      cout << "z= " << z << endl
18     //      << "y= " << y << endl;
19
20     y++;
21     cout << "z= " << z << endl
22         << "y= " << y << endl;
23
24     return 0;
25 }

```

Listing 7. Ejemplo de variables por referencia.

2.8 Asignación de memoria en C++

En el **ANSI C**, se utilizan *malloc*, *calloc* y *free* para asignar y liberar dinámicamente memoria:

```

1  float *f;
2  f = (float *) malloc(sizeof(float));
3  . . .
4  free(f);

```

Se debe indicar el tamaño a través de *sizeof* y utilizar una máscara (*cast*) para designar el tipo de dato apropiado.

En C++, existen dos operadores para asignación y liberación de memoria dinámica: *new* y *delete*.

```
1 float *f;  
2 f= new float;  
3 . . .  
4 delete f;
```

El operador *new* crea automáticamente un área de memoria del tamaño adecuado. Si no se pudo asignar la memoria se regresa un apuntador nulo (*nullptr* ó 0). Nótese que en C++ se trata de operadores que forman parte del lenguaje, no de funciones de biblioteca.

El operador *delete* libera la memoria asignada previamente por *new*. No se debe tratar de liberar memoria previamente liberada o no asignada con *new*.

Es posible hacer asignaciones de memoria con inicialización:

```
int *max= new int (1000);
```

También es posible crear arreglos dinámicamente:

```
1 char *cad;  
2 cad= new char [30];  
3 . . .  
4 delete [] cad;
```

Usar *delete* sin los corchetes para arreglos dinámicos puede no liberar adecuadamente la memoria, sobre todo si son elementos de un tipo definido por el usuario.

Ejemplo 1:

```
1  
2 #include <iostream>  
3  
4 using namespace std;  
5  
6 int main() {  
7     int *p, *q;  
8  
9     p= new int; //asigna memoria
```



```
10
11     if(!p) {
12         cout<<"No se pudo asignar memoria\n";
13         return 0;
14     }
15     *p=100;
16     cout<<endl<< *p<<endl;
17
18     q= new int (123); //asigna memoria
19     cout<<endl<< *q<<endl;
20
21     delete p; //libera memoria
22     //      *p=20;                Uso indebido de pues ya se liberó
23     //      cout<<endl<< *p<<endl; la memoria
24     delete q;
25     return 0;
26 }
```

Listing 8. Ejemplo 1 de asignación de memoria en C++

Ejemplo 2:

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      float *ap, *p=new float (3) ;
8      const int MAX=5;
9      ap= new float [MAX]; //asigna memoria
10     int i;
11     for(i=0; i<MAX; i++)
12         ap[i]=i * *p;
13     for(i=0; i<MAX; i++)
14         cout<<ap[i]<<endl;
15     delete p;
16     delete [] ap;
17     return 0;
18 }
```

Listing 9. Ejemplo 2 de asignación de memoria en C++

Ejemplo 3:

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  typedef struct {
7      int n1,
8          n2,
9          n3;
10 } cPrueba;
11
12 int main() {
13     cPrueba *pr1, *pr2;
14
15     pr1= new cPrueba;
16     pr1->n1=11;
17     pr1->n2=12;
18     pr1->n3=13;
19
20     pr2= new cPrueba(*pr1);
21     delete pr1;
22
23     cout<< pr2->n1<<" "<<pr2->n2 <<" "<<pr2->n3<<endl;
24
25     delete pr2;
26     return 0;
27 }
```

Listing 10. Ejemplo 3 de asignación de memoria en C++

2.9 Plantillas

Cuando las operaciones son idénticas pero requieren de diferentes tipos de datos, podemos usar lo que se conoce como *templates* o plantillas de función.

El término de plantilla es porque el código sirve como base (o plantilla) a diferentes tipos de datos. C++ genera al compilar el código objeto de las funciones para cada tipo de dato involucrado en las llamadas. Esta solución antes se hacía en C usando macros con `#define`, pero no tenían verificación de tipos.

Las definiciones de plantilla se escriben con la palabra clave *template*, con una

lista de parámetros formales entre `<>` . Cada parámetro formal lleva la palabra clave *class*. La instrucción *type* puede también ser usada.

Cada parámetro formal puede ser usado para sustituir a: tipos de datos básicos, estructurados o definidos por el usuario, tipos de los argumentos, tipo de regreso de la función y para variables dentro de la función.

Ejemplo 1:

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  template <class T>
7  T mayor(T x, T y)
8  {
9      return (x > y) ? x : y;
10 }
11
12 int main() {
13     int a=10, b=20, c=0;
14     float x=44.1, y=22.3, z=0 ;
15
16     c=mayor(a, b);
17     z=mayor(x, y);
18     cout<<c<<" "<<z<<endl;
19
20     //      z=mayor(x,b); error no hay mayor( float, int)
21     //      z=mayor(a, y); "" "" "" "" (int, float)
22
23     return 0;
24 }
```

Listing 11. Ejemplo.

Consideraciones:

- Cada parámetro formal debe aparecer en la lista de parámetros de la función al menos una vez.
- No puede repetirse en la definición de la plantilla el nombre de un parámetro formal.
- Tener cuidado al manejar mas de un parámetro en los templates.

Ejemplo 2:

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  template <class T>
7  void desplegaArr(T arr[], const int cont, T pr)
8  {
9      for(int i=0; i<cont; i++)
10         cout<< arr[i] << " ";
11     cout<<endl;
12     cout<<pr<<endl;
13 }
14
15 int main() {
16     const int contEnt=4, contFlot=5, contCar=10;
17     int ent[]={1,2,3,4};
18     float flot[]={1.1, 2.2, 3.3, 4.4, 5.5};
19     char car[]={"Plantilla"};
20
21
22     cout<< "Arreglo de flotantes:\n";
23     desplegaArr(flot, contFlot, (float)3.33);
24
25     cout<< "Arreglo de caracteres:\n";
26     desplegaArr(car, contCar, 'Z');
27
28     cout<< "Arreglo de enteros:\n";
29     desplegaArr(ent, contEnt, 99);
30
31     return 0;
32 }
33
```

Listing 12. Ejemplo 2 de uso de plantillas en C++.

Ejemplo 3:

```
1
2  #include <iostream>
3
```

```
4 using namespace std;
5
6 template <class T, class TT>
7 T mayor(T x, TT y)
8 {
9     return (x > y) ? x : y;
10 };
11
12 int main() {
13
14     int a=10, b=20, c=0;
15     float x=44.1, y=22.3, z=0 ;
16
17     c=mayor(a, b);
18     z=mayor(x, y);
19
20     cout<<c<<" "<<z<<endl;
21     //sin error al aumentar un parámetro formal.
22     z=mayor(x,b);
23     cout<<z<<endl;
24     z=mayor(a,y); //regresa entero pues a es entero (tipo T es entero para
25     cout<<z<<endl; // este llamado.
26
27     z=mayor(y, a);
28     cout<<z<<endl;
29     c=mayor(y, a); //regresa flotante pero la asignación lo corta en entero.
30     cout<<c<<endl;
31
32     return 0;
33 }
```

Listing 13. Ejemplo 3 de uso de plantillas en C++.

2.10 Enumeraciones

Aunque las enumeraciones existen en ANSI C, en ese lenguaje son constantes asociadas al tipo entero; por lo que son una especie de alias hacia estos valores. En C++ una enumeración define realmente un tipo de dato.¹

Las enumeraciones sirven para agrupar un conjunto de elementos dentro de un tipo definido. El manejo de enumeraciones en C++ es el siguiente:

¹Existe otro tipo de enumeración en C++ llamado *enumclass*

Sintaxis

```
enum <nombreEnum> { <elem 1>, <elem 2>, ..., <elem n> };
```

Por lo que para el código anterior, la enumeración sería:

```
enum Temporada { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
```

Ejemplo:

```

1
2  #include <iostream>
3
4  using namespace std;
5
6  enum Temporada { PRIMAVERA, VERANO, OTONO, INVIERNO };
7
8  int main() {
9      Temporada tem;
10     tem=PRIMAVERA;
11     cout<<"Temporada: " << tem<<endl;
12
13     cout<<"\nListado de temporadas:";
14     int t;
15     for(t=PRIMAVERA; t<=INVIERNO; t++)
16         cout<<"Temporada: "<< t<<endl;
17     return 0;
18 }
```

Listing 14. Ejemplo de enumeración en C++.

2.11 Espacios de nombre (*namespaces*)

Los espacios de nombres permiten agrupar entidades que de otro modo tendrían un alcance global. Es preferible tener un alcance de espacio de nombre que un alcance global. Un *namespace* proporciona un alcance para identificadores tales como tipos, funciones, variables, etc. Algunas características²:

- Se pueden tener múltiples bloques con el mismo *namespace*, todas las declaraciones en bloques con el mismo nombre irán al mismo espacio de nombres.

²Más información: [C++ reference: namespace](#)

- Es posible tener espacios de nombre anidados.
- Las declaraciones de espacios de nombre no dan acceso público o privado.

Sintaxis

```
namespace <nombre> {  
    int x, y; // declaraciones de código donde  
              // x , y son declaradas en el alcance de <nombre>  
}
```

Ejemplo:

```
1  
2 #include <iostream>  
3 using namespace std;  
4  
5 namespace ns1 {  
6     int valor() { return 5; }  
7 }  
8 namespace ns2 {  
9     const double x = 100;  
10    double valor() { return 2*x; }  
11 }  
12  
13 int main() {  
14     cout << ns1::valor() << '\n';  
15     cout << ns2::valor() << '\n';  
16     cout << ns2::x << '\n';  
17 }  
18
```

Listing 15. Ejemplo de uso de *namespace*.

2.12 Tipo de dato booleano

Este tipo de dato (*bool*) maneja los valores verdadero o falso (*true* o *false*). Es un tipo de dato y sus literales son además parte de las palabras reservadas de C++. Puede ser declarado como sigue:

```
1 bool b1;
2 b1=true;
3 bool b2 = false;
```

Sin embargo, al final el tipo booleano en C++ sigue siendo equivalente a su uso en el lenguaje C: cero para falso y diferente de cero es verdadero. Por lo que su uso no es obligatorio.

2.13 Uso de cadenas con *string*

En C++ podemos hacer uso de cadenas por medio del tipo *string*. En compiladores actuales es suficiente con incluir la biblioteca *iostream* aunque en realidad se encuentra directamente declarado en *cstring*

Ejemplo:

```
1
2 #include<iostream>
3 using namespace std;
4
5 int main() {
6     string nombre="Juan", apellido="Pérez";
7     cout << "Hola, " << nombre << " " << apellido << endl;
8     cout << "¿Cómo estás?" << endl;
9 }
10
```

Listing 16. Ejemplo de uso de cadenas *string*.

2.14 Ciclo *foreach* en C++

C++ tiene un estilo de bucle *for each* llamado *for (auto element : container)*, el cual se utiliza para recorrer un contenedor de elementos. Este estilo de iteración se introdujo en C++11, presentado en 2011, y es una forma más simple y legible de recorrer un contenedor en comparación con el uso tradicional de un ciclo *for* con un iterador.

Sintaxis

```
for (<tipo_de_dato> <nombre_variable> : <tipo_contenedor>) {  
    <instrucciones>  
}
```

Por ejemplo:

```
1  
2 #include <iostream>  
3  
4 using namespace std;  
5  
6 int main() {  
7     int myArray[] = {1, 2, 3, 4, 5};  
8  
9     // Recorriendo el array con un ciclo for each  
10    for (int element : myArray) {  
11        cout << element << " ";  
12    }  
13    cout << endl;  
14  
15    // Recorriendo el array con un ciclo for each usando auto  
16    for (auto element : myArray) {  
17        cout << element << " ";  
18    }  
19    cout << endl;  
20    return 0;  
21 }
```

Listing 17. Ejemplo de *foreach* en C++.

En este ejemplo se define un arreglo de enteros llamado *myArray* y se inicializa con los valores 1, 2, 3, 4, 5. Luego se utiliza el estilo de ciclo *for each* para recorrer cada elemento del arreglo y se imprime cada elemento en pantalla. El código anterior imprimiría 12345 en pantalla.

Es importante mencionar que el estilo de iteración *for each* sólo es válido para contenedores que soporten el acceso aleatorio, tales como arreglos y vector, entre otros.

Capítulo 3

Introducción a Java

3.1 Origen

Java es un lenguaje de programación orientada a objetos, diseñado dentro de *Sun Microsystems* por James Gosling. Originalmente, se le asignó el nombre de *Oak* y fue un lenguaje pensado para usarse dentro de dispositivos electrodomésticos, que tuvieran la capacidad de comunicarse entre sí. Posteriormente fue reorientado hacia Internet, aprovechando el auge que estaba teniendo en ese momento la red, y lo rebautizaron con el nombre de Java. Es anunciado al público en mayo de 1995 enfocándolo como la solución para el desarrollo de aplicaciones en *web*. Sin embargo, se trata de un lenguaje de propósito general que puede ser usado para la solución de problemas diversos.

Java es un intento serio de resolver simultáneamente los problemas ocasionados por la diversidad y crecimiento de arquitecturas incompatibles, tanto entre máquinas diferentes como entre los diversos sistemas operativos y sistemas de ventanas que funcionaban sobre una misma máquina, añadiendo la dificultad de crear aplicaciones distribuidas en una red como Internet. El interés que generó Java en la industria fue mucho, tal que nunca un lenguaje de programación había sido adoptado tan rápido por la comunidad de desarrolladores. Las principales razones por las que Java es aceptado tan rápido:



- Aprovecha el inicio del auge de la Internet, específicamente del *World Wide Web*.
- Es orientado a objetos, la cual si bien no es tan reciente, estaba en uno de sus mejores momentos, todo mundo quería programar de acuerdo al modelo de objetos.
- Se trataba de un lenguaje que eliminaba algunas de las principales dificultades del lenguaje C/C++, el cuál era uno de los lenguajes dominantes. Se

decía que la ventaja de Java es que es sintácticamente parecido a C++, sin serlo realmente.

- Java era resultado de una investigación con fines comerciales, no era un lenguaje académico como Pascal o creado por un pequeño grupo de personas como C ó C++.

Aunado a esto, las características de diseño de Java, lo hicieron muy atractivo a los programadores.



3.2 Características de diseño

Es un lenguaje de programación de alto nivel, de propósito general, y cuyas características son [?]:

- Simple y familiar.
- Orientado a objetos.
- Independiente de la plataforma
- Portable
- Robusto.
- Seguro.
- Multihilos.

Simple y familiar

Es simple, ya que tanto la estructura léxica como sintáctica del lenguaje es muy sencilla. Además, elimina las características complejas e innecesarias de sus predecesores.

Es familiar al incorporar las mejores características de lenguajes tales como: C/C++, Modula, Beta, CLOS, Dylan, Mesa, Lisp, Smalltalk, Objective-C, y Modula 3.

Orientado a objetos

Es realmente un lenguaje orientado a objetos, todo en Java son objetos:

- No es posible que existan funciones que no pertenezcan a una clase.

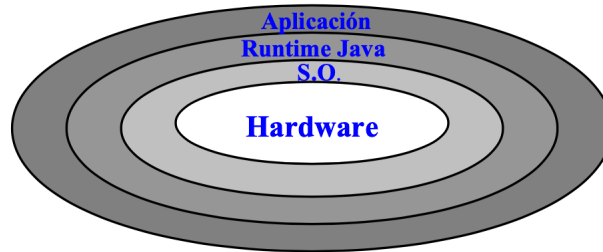


Fig. 3.1. Máquina virtual de Java y su ejecución sobre una plataforma

- La excepción son los tipos de datos primitivos, como números, caracteres y booleanos ¹.

Cumple con los 4 requerimientos de Wegner [?]:

OO = abstracción + clasificación + polimorfismo + herencia

Independiente de la plataforma

La independencia de la plataforma implica que un programa en Java se ejecute sin importar el sistema operativo que se este ejecutando en una máquina en particular. Por ejemplo un programa en C++ compilado para *Windows*, debe ser al menos vuelto a compilar si se quiere ejecutar en Unix; además, posiblemente habrá que ajustar el código que tenga que ver con alguna característica particular de la plataforma, como la interfaz con el usuario.

Java resuelve el problema de la distribución binaria mediante un formato de código binario (*bytecode*) que es independiente del hardware y del sistema operativo gracias a su máquina virtual.

Si el sistema de *runtime* o máquina virtual está disponible para una plataforma específica, entonces una aplicación puede ejecutarse sin necesidad de un trabajo de programación adicional. Ver figura 3.1.

Portable

Una razón por la que los programas en Java son portables es precisamente que el lenguaje es independiente de la plataforma.

Además, la especificación de sus tipos de datos primitivos y sus tamaños, así como el comportamiento de los operadores aritméticos, son estándares en todas las implementaciones de Java. Por lo que por ejemplo, un entero es definido de un tamaño de 4 bytes, y este espacio ocupará en cualquier plataforma, por lo que no tendrá problemas en el manejo de los tipos de datos. En cambio, un entero en C

¹Los puristas objetarían que no es totalmente orientado a objetos. En un sentido estricto *Smalltalk* es un lenguaje “más” puro, ya que ahí hasta los tipos de datos básicos son considerados objetos.

generalmente ocupa 2 bytes, pero en algunas plataformas el entero ocupa 4 bytes, lo que genera problemas a la hora de adaptar un programa de una plataforma a otra.

Robusto

Java se considera un lenguaje robusto y confiable, gracias a:

- **Validación de tipos.** Los objetos de tipos compatibles pueden ser asignados a otros objetos sin necesidad de modificar sus tipos. Los objetos de tipos potencialmente incompatibles requieren un modificador de tipo (*cast*). Si la modificación de tipo es claramente imposible, el compilador lo rechaza y reporta un **error en tiempo de compilación**. Si la modificación resulta legal, el compilador lo permite, pero inserta una **validación en tiempo de ejecución**. Cuando el programa se ejecuta se realiza la validación cada vez que se ejecuta una asignación potencialmente inválida.
- **Control de acceso a variables y métodos.** Los miembros de una clase pueden ser privados, públicos o protegidos². En java una variable privada, es realmente privada. Tanto el compilador como la máquina virtual de Java, controlan el acceso a los miembros de una clase, garantizando así su privacidad.
- **Validación del apuntador Null.** Todos los programas en Java usan apuntadores para referenciar a un objeto. Esto no genera inestabilidad pues una validación del apuntador a *Null* ocurre cada vez que un apuntador deja de referenciar a un objeto. C y C++ por ejemplo, no tienen esta consideración sobre los apuntadores, por lo que es posible estar referenciando a localidades inválidas de la memoria.
- **Límites de un arreglo.** Java verifica en tiempo de ejecución que un programa no use arreglos para tratar de acceder a áreas de memoria que no le pertenecen. De nuevo, C y C++ no tiene esta verificación, lo que permite que un programa se salga del límite mayor y menor de un arreglo.
- **Aritmética de apuntadores.** Aunque todos los objetos se manejan con apuntadores, Java elimina la mayor parte de los errores de manejo de apuntadores porque no soporta la aritmética de apuntadores:
 - No soporta acceso directo a los apuntadores
 - No permite operaciones sobre apuntadores.

²Más adelante en el curso se ahondará en el tema.

- **Manejo de memoria.** Muchos de los errores de la programación se deben a que el programa no libera la memoria que debería liberar, o se libera la misma memoria más de una vez. Java, hace recolección automática de basura, liberando la memoria y evitando la necesidad de que el programador se preocupe por liberar memoria que ya no utilice.

3.3 Diferencias entre Java y C++

Se compara mucho al lenguaje de Java con C++, y esto es lógico debido a la similitud en la sintaxis de ambos lenguajes, por lo que resulta necesario resaltar las diferencias principales que existen entre estos. Java a diferencia de C++:

- **No tiene aritmética de apuntadores.** Java si tiene apuntadores, sin embargo no permite la manipulación directa de las direcciones de memoria. No se proporcionan operadores para el manejo de los apuntadores pues no se considera responsabilidad del programador.
- **No permite funciones con ámbito global.** Toda operación debe estar asociada a una clase, por lo que el ámbito de la función esta limitado al ámbito de la clase.
- **Elimina la instrucción *goto*.** La instrucción *goto* se considera obsoleta, ya que es una herencia de la época de la programación no estructurada. Sin embargo, esta definida como palabra reservada para restringir su uso.
- **Las cadenas no terminan con `'\0'`.** Las cadenas en C y C++ terminan con `'\0'` que corresponde al valor en ASCII 0, ya que en estos lenguajes no existe la cadena como un tipo de dato estándar y se construye a partir de arreglos de caracteres. En Java una cadena es un objeto de la clase *String* y no necesita ese carácter para indicar su finalización.
- **No maneja macros.** Una macro es declarada en C y C++ a través de la instrucción *#define*, la cual es tratada por el preprocesador.
- **No soporta un preprocesador.** Una de las razones por las cuales no maneja macros Java es precisamente porque no tiene un preprocesador que prepare el programa previo a la compilación.
- **El tipo *char* contiene 16 bits.** Un carácter en Java utiliza 16 bits en lugar de 8 para poder soportar UNICODE en lugar de ASCII, lo que permite la representación de múltiples símbolos.
- **Java soporta múltiples hilos de ejecución.** Los múltiples hilos de ejecución o multihilos permiten un fácil manejo de programación concurrente. Otros lenguajes dependen de la plataforma para implementar concurrencia.

- **Todas las condiciones en Java deben tener como resultado un tipo booleano.** Debido a que en Java los resultados de las expresiones son dados bajo este tipo de dato. Mientras que en C y C++ se considera a un valor de cero como falso y no cero como verdadero.
- **Java no soporta el operador *sizeof*.** Este operador permite en C y C++ obtener el tamaño de una estructura de datos. En Java esto no es necesario ya que cada objeto “sabe” el espacio que ocupa en memoria.
- **No tiene herencia múltiple.** Java solo cuenta con herencia simple, con lo que pierde ciertas capacidades de generalización que son subsanadas a través del uso de interfaces. El equipo de desarrollo de Java explica que esto simplifica el lenguaje y evita la ambigüedad natural generada por la herencia múltiple.
- **No tiene liberación de memoria explícita (*delete* y *free()*).** En Java no es necesario liberar la memoria ocupada, ya que cuenta con un recolector de basura ³ responsable de ir liberando cada determinado tiempo los recursos de memoria que ya no se estén ocupando.

Además:

- No contiene estructuras y uniones (*struct* y *union*).
- No contiene tipos de datos sin signo.
- No permite alias (*typedef*).
- No tiene conversión automática de tipos compatibles.

3.4 Archivos .java y .class

En Java el código fuente se almacena en archivos con extensión .java, mientras que el *bytecode* o código compilado se almacena en archivos .class. El compilador de Java crea un archivo .class por cada declaración de clase que encuentra en el archivo .java.

Un archivo de código fuente debe tener solo una clase principal, y ésta debe tener exactamente el mismo nombre que el del archivo .java. Por ejemplo, si tengo una clase que se llama *Alumno*, el archivo de código fuente se llamará *Alumno.java*. Al compilar, el archivo resultante será *Alumno.class*.

³Garbage Collector

3.5 Programas generados con java

Existen dos tipos principales de programas en Java ⁴:

Por un lado están las aplicaciones, las cuales son programas *standalone*, escritos en Java y ejecutados por un intérprete del código de bytes desde la línea de comandos del sistema.

Por otra parte, las *Aplicaciones Gráficas*, que son programas que utilizan bibliotecas como JavaFX o Swing para crear interfaces de usuario ricas. Anteriormente se utilizaban *Applets*, pero han sido depreciados y eliminados en versiones modernas del JDK.

En términos del código fuente las diferencias entre una aplicación de consola y una gráfica son:

- Una aplicación de consola debe definir una clase que contenga el método *main()*, que controla su ejecución.
- Una aplicación gráfica con JavaFX hereda de la clase *Application* y sobrescribe el método *start()*.

3.6 El Java Developer's Kit

La herramienta básica para empezar a desarrollar aplicaciones en Java es el JDK (*Java Development Kit*) o Kit de Desarrollo Java, que consiste esencialmente, en un compilador y un intérprete (JVM⁵) para la línea de comandos. No dispone de un entorno de desarrollo integrado (IDE), pero es suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones⁶.

Los principales programas del *Java Development Kit*:

- **javac**. Es el compilador en línea del JDK.
- **java**. Es la máquina virtual para aplicaciones de Java.
- **javadoc**. Generador de documentación.

⁴Se presenta la división clásica de los programas de Java, aunque existen algunas otras opciones no son relevantes en este curso.

⁵*Java Virtual Machine*

⁶Este kit de desarrollo es gratuito y puede obtenerse de la dirección proporcionada al final de este documento. Independientemente del IDE que se use, el jdk debe estar instalado para poder compilar código Java.

Compilación

Utilizando el JDK, los programas se compilan desde el símbolo del sistema con el compilador `javac`.

Ejemplo:

```
C:  
MisProgramas> javac MiClase.java
```

Normalmente se compila como se ha mostrado en el ejemplo anterior. Sin embargo, el compilador proporciona diversas opciones a través de modificadores que se agregan en la línea de comandos.

Sintaxis
<code>javac [opciones] <archivo1.java></code>

donde opciones puede ser:

- `-classpath < ruta >` Indica donde buscar los archivos de clase de Java
- `-d < directorio >` Indica el directorio destino para los archivos `.class`
- `-g` Habilita la generación de tablas de depuración.
- `-nowarn` Deshabilita los mensajes del compilador.
- `-O` Optimiza el código, generando en línea los métodos estáticos, finales y privados.
- `-verbose` Indica cuál archivo fuente se esta compilando.

3.7 “Hola Mundo”

Para no ir en contra de la tradición al comenzar a utilizar un lenguaje, los primeros ejemplos son precisamente dos programas muy simples que lo único que van a hacer es desplegar el mensaje “Hola Mundo”.

Hola mundo básico en Java El primero es una aplicación que va a ser interpretado posteriormente por la máquina virtual:

```
1
2 public class HolaMundo {
3     public static void main(String args[]) {
4         System.out.println("¡Hola, Mundo!");
5     }
6 }
```

Listing 18. Hola, Mundo en Java.

Para los que han programado en C ó C++, notarán ya ciertas similitudes. Lo importante aquí es que una aplicación siempre requiere de un método *main*, este tiene un solo argumento (*String args[]*), a través del cual recibe información de los argumentos de la línea de comandos, pero la diferencia con los lenguajes C/C++ es que este método depende de una clase, en este caso la clase *HolaMundo*. Este programa es compilado en al jdk⁷:

```
%javac HolaMundo.java
```

con lo que, si el programa no manda errores, se obtendrá el archivo *HolaMundo.class*.

En Eclipse, al grabar automáticamente el programa se compilará (si la opción *Build Automatically* está activada). De hecho, algunos errores se van notificando, si los hay, conforme se va escribiendo el código en el editor.

Hola mundo básico en C++

En C++ no estamos obligados a usar clases, por lo que un “Hola mundo” en C++ - aunque no en objetos – podría quedar de la siguiente forma:

```
1
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     cout << "Hola Mundo!" << endl;
8     return 0;
9 }
```

Listing 19. Hola, Mundo en C++.

⁷Se asume que el jdk se encuentra instalado y que el PATH tiene indicado el directorio bin del jdk para que encuentre el programa javac. También es recomendable añadir nuestro directorio de programas de java a una variable de ambiente llamada CLASSPATH.

Ejecución

Para ejecutar una aplicación usamos la máquina virtual proporcionada por el *jdk*, proporcionando el nombre de la clase:

```
% java HolaMundo
```

A partir de la versión 11 del *jdk*⁸, podemos ejecutar directamente ejemplos pequeños de código java. Se compila y ejecuta, sin generar el código *.class* correspondiente:

```
% java HolaMundo.java
```

3.8 Fundamentos del Lenguaje Java

En esta sección se hablará de cómo está constituido el lenguaje, sus instrucciones, tipos de datos, entre otras características. Antes de comenzar a hacer programación orientada a objetos.

3.8.1 Comentarios

Los comentarios en los programas fuente son muy importantes en cualquier lenguaje. Sirven para aumentar la facilidad de comprensión del código y para recordar ciertas cosas sobre el mismo. Son porciones del programa fuente que el compilador omite, y, por tanto, no ocuparán espacio en el archivo de clase.

Existen tres tipos de comentarios en Java:

- Si el comentario que se desea escribir es de una sola línea, basta con poner dos barras inclinadas `//`. Por ejemplo:

```
1 for (i=0; i<20;i++) // comentario de ciclo {  
2     System.out.println("Adiós");  
3 }
```

No puede ponerse código después de un comentario introducido por `//` en la misma línea, ya que desde la aparición de las dos barras inclinadas `//` hasta el final de la línea es considerado como comentario e ignorado por el compilador.

- Si un comentario debe ocupar más de una línea, hay que anteponerle `/*` y al final `*/`. Por ejemplo:

```
1  /* Esto es un
2  comentario que
3  ocupa tres líneas */
```

- Existe otro tipo de comentario que sirve para generar documentación automáticamente en formato HTML mediante la herramienta javadoc. Puede ocupar varias líneas y se inicia con `/**` para terminar con `*/`. Para mas información ver: <http://java.sun.com/j2se/javadoc/>

3.8.2 Tipos de datos

En Java existen dos tipos principales de datos:

1. Tipos de datos **simples**.
2. **Referencias** a objetos.

Los tipos de datos simples son aquellos que pueden utilizarse directamente en un programa, sin necesidad del uso de clases. Estos tipos son:

- byte
- short
- int
- long
- float
- double
- char
- boolean

El segundo tipo está formado por todos los demás. Se les llama referencias porque en realidad lo que se almacena en los mismos son punteros a áreas de memoria donde se encuentran almacenadas las estructuras de datos que los soportan. Dentro de este grupo se encuentran las clases (objetos) y también se incluyen las interfaces, las colecciones (como *ArrayList*) y las cadenas o *Strings*.

Pueden realizarse conversiones entre los distintos tipos de datos (incluso entre simples y referenciales), bien de forma implícita o de forma explícita.

Tipo	Descripción	Formato	Longitud	Rango
byte	byte	C-2 ¹⁰	1 byte	- 128 ... 127
short	entero corto	C-2	2 bytes	- 32.768 ... 32.767
int	entero	C-2	4 bytes	- 2.147.483.648 ... 2.147.483.647
long	entero largo	C-2	8 bytes	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	real en coma flotante de precisión simple	IEEE 754	32 bits	$3,4 * 10_{-38} \dots 3,4 * 10_{38}$
double	real en coma flotante de precisión doble	IEEE 754	64 bits	$1,7 * 10_{-308} \dots 1,7 * 10_{308}$
char	Carácter	Unicode	2 bytes	0 ... 65.535
boolean	Lógico		1 bit	true / false

Cuadro 3.1. Tipos de datos simples en Java

Tipos de datos simples

Los tipos de datos simples en Java tienen las siguientes características:

No existen más datos simples en Java. Incluso éstos que se enumeran pueden ser remplazados por clases equivalentes (*Integer*, *Double*, *Byte*, etc.), con la ventaja de que es posible tratarlos como si fueran objetos en lugar de datos simples.

A diferencia de otros lenguajes de programación como C, en Java los tipos de datos simples no dependen de la plataforma ni del sistema operativo. Un entero de tipo *int* siempre tendrá 4 bytes, por lo que no tendremos resultados inesperados al migrar un programa de un sistema operativo a otro.

Eso sí, Java no realiza una comprobación de los rangos. Por ejemplo: si a una variable de tipo *short* con el valor 32.767 se le suma 1, el resultado será -32.768 y no se producirá ningún error de ejecución.

Los valores que pueden asignarse a variables y que pueden ser utilizados en expresiones directamente reciben el nombre de literales.

Referencias a objetos

El resto de tipos de datos que no son simples, son considerados referencias. Estos tipos son básicamente apuntadores a las instancias de las clases, en las que se basa la programación orientada a objetos.

Al declarar una variable de objeto perteneciente a una determinada clase, se indica que ese identificador de referencia tiene la capacidad de apuntar a un objeto del tipo al que pertenece la variable. El momento en el que se realiza la reserva física del espacio de memoria es cuando se instancia el objeto realizando la llamada a su constructor, y no en el momento de la declaración.

Existe un tipo referencial especial nominado por la palabra reservada *null* que puede ser asignado a cualquier variable de cualquier clase y que indica que el puntero no tiene referencia a ninguna zona de memoria (el objeto no está inicializado).

3.8.3 Identificadores

Los identificadores son los nombres que se les da a las variables, clases, interfaces, atributos y métodos de un programa.

Existen algunas reglas básicas para nombrar a los identificadores:

1. Java hace distinción entre mayúsculas y minúsculas, por lo tanto, nombres o identificadores como *var1*, *Var1* y *VAR1* son distintos.
2. Pueden estar formados por cualquiera de los caracteres del código *Unicode*, por lo tanto, se pueden declarar variables con el nombre: *añoDeCreación*, *raïm*, etc.
3. El primer carácter no puede ser un dígito numérico y no pueden utilizarse espacios en blanco ni símbolos coincidentes con operadores.
4. No puede ser una palabra reservada del lenguaje ni los valores lógicos *true* o *false*.
5. No pueden ser iguales a otro identificador declarado en el mismo ámbito.
6. Por convención, los nombres de las variables y los métodos deberían empezar por una letra minúscula y los de las clases por mayúscula.

Además, si el identificador está formado por varias palabras, la primera se escribe en minúsculas (excepto para las clases e interfaces) y el resto de palabras se hace empezar por mayúscula (por ejemplo: *añoDeCreación*). Las constantes se escriben en mayúsculas, por ejemplo *MÁXIMO*.

Esta última regla no es obligatoria, pero es conveniente ya que ayuda al proceso de codificación de un programa, así como a su legibilidad. Es más sencillo distinguir entre clases y métodos, variables o constantes.

3.8.4 Variables

La declaración de una variable se realiza de la misma forma que en C/C++. Siempre contiene el nombre (identificador de la variable) y el tipo de dato al que pertenece. El ámbito de la variable depende de la localización en el programa donde es declarada.

Ejemplo:

```
int x;
```

Las variables pueden ser inicializadas en el momento de su declaración, siempre que el valor que se les asigne coincida con el tipo de dato de la variable.

Ejemplo:

```
int x = 0;
```

Ámbito de una variable

El ámbito de una variable es la porción de programa donde dicha variable es visible para el código del programa y, por tanto, referenciable. El ámbito de una variable depende del lugar del programa donde es declarada, pudiendo pertenecer a cuatro categorías distintas.

- Variable local.
- Atributo.
- Parámetro de un método.
- Parámetro de un manejador de excepciones¹¹.

Como puede observarse, no existen las variables globales. La utilización de variables globales es considerada peligrosa, ya que podría ser modificada en cualquier parte del programa y por cualquier procedimiento. A la hora de utilizarlas hay que buscar dónde están declaradas para conocerlas y dónde son modificadas para evitar sorpresas en los valores que pueden contener.

Los ámbitos de las variables u objetos en Java siguen los criterios “clásicos”, al igual que en la mayoría de los lenguajes de programación como Pascal, C++, etc.

Si una variable que **no es local** no ha sido inicializada, tiene un valor asignado por defecto. Este valor es, para las variables referencias a objetos, el valor *null*. Para las variables de tipo numérico, el valor por defecto es cero, las variables de tipo *char*, el valor ‘u0000’ y las variables de tipo *boolean*, el valor *false*.

Variables locales Una variable local se declara dentro del cuerpo de un método de una clase y es visible únicamente dentro de dicho método. Se puede declarar en cualquier lugar del cuerpo, incluso después de instrucciones ejecutables, aunque es una buena costumbre declararlas justo al principio.

3.8.5 Operadores

Los operadores son partes indispensables en la construcción de expresiones. Existen muchas definiciones técnicas para el término expresión. Puede decirse que una expresión es una combinación de operandos ligados mediante operadores.

Los operandos pueden ser variables, constantes, funciones, literales, etc. y los operadores se comentarán a continuación.

Operador	Formato	Descripción
+	op1 + op2	Suma aritmética de dos operandos
-	op1 - op2	Resta aritmética de dos operandos
-op1		Cambio de signo
*	op1 * op2	Multiplicación de dos operandos
/	op1 / op2	División entera de dos operandos
%	op1 % op2	Resto de la división entera (o módulo)
++	++op1	Incremento unitario
	op1++	
--	--op1	Decremento unitario
	op1--	

Cuadro 3.2. Operadores aritméticos Java

Operadores aritméticos:

El operador - puede utilizarse en su versión unaria (- op1) y la operación que realiza es la de invertir el signo del operando.

Como en C/C++, los operadores unarios ++ y -- realizan un incremento y un decremento respectivamente. Estos operadores admiten notación prefija y postfija. Ver Cuadro 3.2

- ++op1: En primer lugar realiza un incremento (en una unidad) de *op1* y después ejecuta la instrucción en la cual está inmerso.
- op1++: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un incremento (en una unidad) de *op1*.
- --op1: En primer lugar realiza un decremento (en una unidad) de *op1* y después ejecuta la instrucción en la cual está inmerso. Visión General y elementos básicos del lenguaje.
- op1--: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un decremento (en una unidad) de *op1*.

La diferencia entre la notación prefija y la postfija no tiene importancia en expresiones en las que únicamente existe dicha operación:

```

1 ++contador; //es equivalente a: contador++;
2 --contador; //contador--;

```

¹¹Este se tocará en otra etapa del curso, al hablar de manejo de excepciones.

Operador	Formato	Descripción
>	op1 > op2	Devuelve <i>true</i> si op1 es mayor que op2
<	op1 < op2	Devuelve <i>true</i> si op1 es menor que op2
>=	op1 >= op2	Devuelve <i>true</i> si op1 es mayor o igual que op2
<=	op1 <= op2	Devuelve <i>true</i> si op1 es menor o igual que op2
==	op1 == op2	Devuelve <i>true</i> si op1 es igual a op2
!=	op1 != op2	Devuelve <i>true</i> si op1 es distinto de op2

Cuadro 3.3. Operadores relacionales en Java

La diferencia es apreciable en instrucciones en las cuáles están incluidas otras operaciones.

Ejemplo:

```

1 a = 1; a = 1;
2 b = 2 + a++; b = 2 + ++a;

```

En el primer caso, después de las operaciones, b tendrá el valor 3 y al valor 2. En el segundo caso, después de las operaciones, b tendrá el valor 4 y al valor 2.

Operadores relacionales:

Los operadores relacionales actúan sobre valores enteros, reales y caracteres; y devuelven un valor del tipo booleano (*true* o *false*). Ver Cuadro 3.3

Ejemplo:

```

1
2 public class Relacional {
3     public static void main(String arg[]) {
4         double op1, op2;
5         op1=1.34;
6         op2=1.35;
7         System.out.println("op1="+op1+" op2="+op2);
8         System.out.println("op1>op2 = "+(op1>op2));
9         System.out.println("op1<op2 = "+(op1<op2));
10        System.out.println("op1==op2 = "+(op1==op2));
11        System.out.println("op1!=op2 = "+(op1!=op2));
12        char op3, op4;
13        op3='a'; op4='b';
14        System.out.println("'a'>'b' = "+(op3>op4));

```

Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve <i>true</i> si son ciertos op1 y op2
	op1 op2	O lógico. Devuelve <i>true</i> si son ciertos op1 o op2
!	!op1	Negación lógica. Devuelve <i>true</i> si es falso op1.

Cuadro 3.4. Operadores lógicos en Java

```

15     }
16 }

```

Listing 20. Ejemplo de operadores relacionales en Java.

Operadores lógicos:

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso. Ver Cuadro 3.4

[Ejemplo:](#)

```

1
2 public class Bool {
3     public static void main ( String argumentos[] ) {
4         boolean a=true;
5         boolean b=true;
6         boolean c=false;
7         boolean d=false;
8         System.out.println("true Y true = " + (a && b) );
9         System.out.println("true Y false = " + (a && c) );
10        System.out.println("false Y false = " + (c && d) );
11        System.out.println("true O true = " + (a || b) );
12        System.out.println("true O false = " + (a || c) );
13        System.out.println("false O false = " + (c || d) );
14        System.out.println("NO true = " + !a);
15        System.out.println("NO false = " + !c);
16        System.out.println("(3 > 4) Y true = " + ((3 >4) && a) );
17    }
18 }

```

Listing 21. Ejemplo de operadores lógicos en Java.

Operadores de asignación:

El operador de asignación es el símbolo igual (=).

Operador	Formato	Equivalencia
$+$ =	$op1 + = op2$	$op1 = op1 + op2$
$-$ =	$op1 - = op2$	$op1 = op1 - op2$
$*$ =	$op1 * = op2$	$op1 = op1 * op2$
$/$ =	$op1 / = op2$	$op1 = op1 / op2$
$\%$ =	$op1 \% = op2$	$op1 = op1 \% op2$

Cuadro 3.5. Operadores de asignación en Java

```
op1 = Expresión;
```

Asigna el resultado de evaluar la expresión de la derecha a *op1*.

Además del operador de asignación existen unas abreviaturas, como en C/C++, cuando el operando que aparece a la izquierda del símbolo de asignación también aparece a la derecha del mismo. Ver Cuadro 3.5

Precedencia de operadores en Java

La precedencia (ver Cuadro 3.6) indica el orden en que es resuelta una expresión, la siguiente lista muestra primero los operadores de mayor precedencia.

3.8.6 Valores literales

A la hora de tratar con valores de los tipos de datos simples (y *Strings*) se utiliza lo que se denomina “literales”. Los literales son elementos que sirven para representar un valor en el código fuente del programa.

En Java existen literales para los siguientes tipos de datos:

- Lógicos (*boolean*).
- Carácter (*char*).
- Enteros (*byte*, *short*, *int* y *long*).
- Reales (*double* y *float*).
- Cadenas de caracteres (*String*).

Tipo de operador	Operadores
Operadores postfijos	<code>[].(parenthesis)</code>
Operadores unarios	<code>++expr --expr ~expr !</code>
Creación o conversión de tipo	<code>new(tipo)expr</code>
Multiplicación y división	<code>*/%</code>
Suma y resta	<code>+ -</code>
Desplazamiento de bits	<code>~></code>
Relacionales	<code><><=>=</code>
Igualdad y desigualdad	<code>==!=</code>
AND a nivel de bits	<code>&</code>
OR a nivel de bits	<code> </code>
AND lógico	<code>&&</code>
OR lógico	<code> </code>
Condicional terciaria	<code>? :</code>
Asignación	<code>= + = - = * = / = % = & = = ~ = ' = > =</code>

Cuadro 3.6. Precedencia de operadores en Java

Literales lógicos

Son únicamente dos: las palabras reservadas *true* y *false*.

Ejemplo:

```
boolean activado = false;
```

Literales de tipo entero

Son *byte*, *short*, *int* y *long* pueden expresarse en decimal (base 10), octal (base 8) o hexadecimal (base 16). Además, puede añadirse al final del mismo la letra L para indicar que el entero es considerado como *long* (64 bits).

Literales de tipo real

Los literales de tipo real sirven para indicar valores float o double. A diferencia de los literales de tipo entero, no pueden expresarse en octal o hexadecimal.

Existen dos formatos de representación: mediante su parte entera, el punto decimal (`.`) y la parte fraccionaria; o mediante notación exponencial o científica:

Ejemplos equivalentes:

```
3.1415
0.31415e1
.31415e1
0.031415E+2
.031415e2
```

```
314.15e-2  
31415E-4
```

Al igual que los literales que representan enteros, se puede poner una letra como sufijo. Esta letra puede ser una F o una D (mayúscula o minúscula indistintamente).

- F Trata el literal como de tipo *float*.
- D Trata el literal como de tipo *double*.

Ejemplo:

```
3.1415F  
.031415d
```

Literales de tipo carácter

Los literales de tipo carácter se representan siempre entre comillas simples. Entre las comillas simples puede aparecer:

- Un símbolo (letra) siempre que el carácter esté asociado a un código *Unicode*. Ejemplos: 'a' , 'B' , '{' , 'ñ' , 'á' .
- Una “secuencia de escape”. Las secuencias de escape son combinaciones del símbolo seguido de una letra, y sirven para representar caracteres que no tienen una equivalencia en forma de símbolo.

Las posibles secuencias de escape se pueden ver en el Cuadro 3.7

Literales de tipo *String*

Los *Strings* o cadenas de caracteres no forman parte de los tipos de datos elementales en Java, sino que son instanciados a partir de la clase *java.lang.String*, pero aceptan su inicialización a partir de literales de este tipo.

Un literal de tipo *String* va encerrado entre comillas dobles (") y debe estar incluido completamente en una sola línea del programa fuente (no puede dividirse en varias líneas). Entre las comillas dobles puede incluirse cualquier carácter del código *Unicode* (o su código precedido del carácter \) además de las secuencias de escape vistas anteriormente en los literales de tipo carácter. Así, por ejemplo, para incluir un cambio de línea dentro de un literal de tipo *String* deberá hacerse mediante la secuencia de escape \n :

Ejemplo:

Secuencia de escape	Significado
\'	Comilla simple.
\"	Comillas dobles.
\\	Barra invertida.
\b	Backspace (Borrar hacia atrás).
\n	Cambio de línea.
\f	Form feed.
\r	Retorno de carro.
\t	Tabulador.

Cuadro 3.7. Secuencias de escape en Java

```

1 System.out.println("Primera línea \n Segunda línea del string\n");
2 System.out.println("Hol\u0061");

```

La visualización del *String* anterior mediante *println()* produciría la siguiente salida por pantalla:

```

Primera línea
Segunda línea del string
Hola

```

La forma de incluir los caracteres: comillas dobles (") y barra invertida (\) es mediante las secuencias de escape \" y \\ respectivamente (o mediante su código *Unicode* precedido de \).

Si la cadena es demasiado larga y debe dividirse en varias líneas en el código fuente, o simplemente concatenar varias cadenas, puede utilizarse el operador de concatenación de *strings* + .de la siguiente forma:

```

''Este String es demasiado largo para estar en una línea'' +
''del código fuente y se ha dividido en dos.''

```

3.8.7 Estructuras de control

Las estructuras de control son construcciones definidas a partir de palabras reservadas del lenguaje que permiten modificar el flujo de ejecución de un programa. De este modo, pueden crearse construcciones de decisión y ciclos de repetición de bloques de instrucciones.

Hay que señalar que, como en C/C++, un bloque de instrucciones se encontrará encerrado mediante llaves {...} si existe más de una instrucción.

Estructuras condicionales

Las estructuras condicionales o de decisión son construcciones que permiten alterar el flujo secuencial de un programa, de forma que en función de una condición o el valor de una expresión, el mismo pueda ser desviado en una u otra alternativa de código.

Las estructuras condicionales disponibles en Java son:

- Estructura if-else.
- Estructura switch.

if-else

Sintaxis
Forma simple:
<pre>if (<expresión>) <Bloque instrucciones></pre>

El bloque de instrucciones se ejecuta si, y sólo si, la expresión (que debe ser lógica) se evalúa a verdadero, es decir, se cumple una determinada condición.

Ejemplo:

```
1 if (cont == 0)  
2     System.out.println("he llegado a cero");
```

La instrucción `System.out.println("he llegado a cero");` sólo se ejecuta en el caso de que `cont` contenga el valor cero.

Sintaxis

Forma bicondicional:

```
if (<expresión>
    <Bloque instrucciones 1>
else
    <Bloque instrucciones 2>
```

El bloque de instrucciones 1 se ejecuta si, y sólo si, la expresión se evalúa como verdadero. Y en caso contrario, si la expresión se evalúa como falso, se ejecuta el bloque de instrucciones 2.

Ejemplo:

```
1 if (cont == 0)
2     System.out.println("he llegado a cero");
3 else
4     System.out.println("no he llegado a cero");
```

En Java, como en C/C++ y a diferencia de otros lenguajes de programación, en el caso de que el bloque de instrucciones conste de una sola instrucción no necesita ser encerrado en un bloque.

instrucción switch

Sintaxis

```
switch (<expresión>) {
case <valor1>: <instrucciones1>;
case <valor2>: <instrucciones2>;
...
case <valorN>: <instruccionesN>;
}
```

En este caso, a diferencia del *if*, si *< instrucciones1 >*, *< instrucciones2 >* ó *< instruccionesN >* están formados por un bloque de instrucciones sencillas, no es necesario encerrarlas mediante las llaves (...).

En primer lugar se evalúa la expresión cuyo resultado puede ser un valor de cualquier tipo. El programa comprueba el primer valor (*valor1*). En el caso de que el valor resultado de la expresión coincida con *valor1*, se ejecutará el bloque *< instrucciones1 >*. Pero también se ejecutarían el bloque *< instrucciones2 >* ... *< instruccionesN >* hasta encontrarse con la palabra reservada *break*. Por lo que comúnmente se añade una instrucción *break* al final de cada caso del *switch*.

Ejemplo:

```
1  switch (<expresión>) {  
2  
3      case <valor1>: <instrucciones1>;  
4                      break;  
5      case <valor2>: <instrucciones2>;  
6                      break;  
7      ...  
8      case <valorN>: <instruccionesN>;  
9  }
```

Si el resultado de la expresión no coincide con *< valor1 >*, evidentemente no se ejecutarían *< instrucciones1 >*, se comprobaría la coincidencia con *< valor2 >* y así sucesivamente hasta encontrar un valor que coincida o llegar al final de la construcción *switch*. En caso de que no exista ningún valor que coincida con el de la expresión, no se ejecuta ninguna acción.

Ejemplo:

```
1  
2  public class DiaSemana {  
3      public static void main(String argumentos[]) {  
4          int dia;  
5          if (argumentos.length<1) {  
6              System.out.println("Uso: DiaSemana num");  
7              System.out.println("Donde num= nº entre 1 y 7");  
8          }  
9          else {  
10             dia=Integer.valueOf(argumentos[0]);  
11             // también: dia=Integer.parseInt(argumentos[0]);  
12             switch (dia) {  
13                 case 1: System.out.println("Lunes");
```

```
14         break;
15         case 2: System.out.println("Martes");
16         break;
17         case 3: System.out.println("Miércoles");
18         break;
19         case 4: System.out.println("Jueves");
20         break;
21         case 5: System.out.println("Viernes");
22         break;
23         case 6: System.out.println("Sábado");
24         break;
25         case 7: System.out.println("Domingo");
26     }
27 }
28 }
29 }
```

Listing 22. Ejemplo de uso de switch en Java.

Nótese que en el caso de que se introduzca un valor no comprendido entre 1 y 7, no se realizará ninguna acción. Esto puede corregirse agregando la opción por omisión:

```
default: instruccionesPorDefecto;
```

donde la palabra reservada *default*, sustituye a *case < expr >* para ejecutar el conjunto de instrucciones definido en caso de que no coincida con ningún otro caso.

Expresión switch

En la versión de **Java 12 y posteriores**, se introdujo una nueva característica llamada *expresión switch*, que permite un uso más conciso del switch. El estilo sería el siguiente.

Sintaxis

```
int resultado = switch (expresion) {  
    case valor1 -> {  
        // Código a ejecutar si la expresión es igual a valor1  
        yield resultado1; // Opcional: valor a devolver  
    }  
    case valor2 -> {  
        // Código a ejecutar si la expresión es igual a valor2  
        yield resultado2; // Opcional: valor a devolver  
    }  
    // Otros casos aquí  
    default -> {  
        // Código a ejecutar si ninguno de los casos anteriores se cumple  
        yield resultadoDefault; // Opcional: valor a devolver  
    }  
};
```

Esta sintaxis permite un código más limpio y expresivo y también es capaz de devolver un valor en función del caso que coincida.

```
1 public class DiaSemana {  
2     public static void main(String argumentos[]) {  
3         int dia;  
4         if (argumentos.length < 1) {  
5             System.out.println("Uso: DiaSemana num");  
6             System.out.println("Donde num= nº entre 1 y 7");  
7         } else {  
8             dia = Integer.valueOf(argumentos[0]);  
9             // También: dia = Integer.parseInt(argumentos[0]);  
10            String diaDeLaSemana = switch (dia) {  
11                case 1 -> "Lunes";  
12                case 2 -> "Martes";  
13                case 3 -> "Miércoles";  
14                case 4 -> "Jueves";  
15                case 5 -> "Viernes";  
16                case 6 -> "Sábado";  
17                case 7 -> "Domingo";
```

```
18         default -> "Día no válido";
19     };
20     System.out.println(diaDeLaSemana);
21 }
22 }
23 }
```

Listing 23. Ejemplo de uso de la **expresión** switch en Java.

Ciclos

Los ciclos o iteraciones son estructuras de repetición. Bloques de instrucciones que se repiten un número de veces **mientras** se cumpla una condición o **hasta** que se cumpla una condición.

Existen tres construcciones para estas estructuras de repetición:

- Ciclo for.
- Ciclo do-while.
- Ciclo while.

Como regla general puede decirse que se utilizará el ciclo for cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones. Se utilizará el ciclo *do-while* cuando no se conoce exactamente el número de veces que se ejecutará el ciclo pero se sabe que por lo menos se ha de ejecutar una. Se utilizará el ciclo *while* cuando es posible que no deba ejecutarse ninguna vez. Con mayor o menor esfuerzo, puede utilizarse cualquiera de ellas indistintamente.

Ciclo for

Sintaxis

```
for (<inicialización> ; <condición> ; <incremento>)
    <bloque instrucciones>
```

- La cláusula inicialización es una instrucción que se ejecuta una sola vez al inicio del ciclo, normalmente para inicializar un contador.

- La cláusula condición es una expresión lógica, que se evalúa al inicio de cada nueva iteración del ciclo. En el momento en que dicha expresión se evalúe a falso, se dejará de ejecutar el ciclo y el control del programa pasará a la siguiente instrucción (a continuación del ciclo *for*).
- La cláusula incremento es una instrucción que se ejecuta en cada iteración del ciclo como si fuera la última instrucción dentro del bloque de instrucciones. Generalmente se trata de una instrucción de incremento o decremento de alguna variable.

Cualquiera de estas tres cláusulas puede estar vacía, aunque siempre hay que poner los puntos y coma (;).

El siguiente programa muestra en pantalla la serie de *Fibonacci* hasta el término que se indique al programa como argumento en la línea de comandos. Siempre se mostrarán, por lo menos, los dos primeros términos

Ejemplo:

```
1 // siempre se mostrarán, por lo menos, los dos primeros //términos
2 public class Fibonacci {
3     public static void main(String argumentos[]) {
4         int numTerm,v1=1,v2=1,aux,cont;
5         if (argumentos.length<1) {
6             System.out.println("Uso: Fibonacci num");
7             System.out.println("Donde num = n° de términos");
8         }
9         else {
10            numTerm=Integer.valueOf(argumentos[0]);
11            System.out.print("1,1");
12            for (cont=2;cont<numTerm;cont++) {
13                aux=v2;
14                v2+=v1;
15                v1=aux;
16                System.out.print(", "+v2);
17            }
18            System.out.println();
19        }
20    }
21 }
```

Listing 24. Ejemplo Fibonacci con ciclo *for*.

Ciclo do-while

Sintaxis

```
do
    <bloque instrucciones>
while (<Expresión>);
```

En este tipo de ciclo, el bloque instrucciones se ejecuta siempre una vez por lo menos, y el bloque de instrucciones se ejecutará mientras *< Expresion >* se evalúe como verdadero. Por lo tanto, entre las instrucciones que se repiten deberá existir alguna que, en algún momento, haga que la expresión se evalúe como falso, de lo contrario el ciclo sería infinito.

Ejemplo:

```
1 //El mismo que antes (Fibonacci).
2 public class Fibonacci2 {
3     public static void main(String argumentos[]) {
4         int numTerm,v1=0,v2=1,aux,cont=1;
5         if (argumentos.length<1) {
6             System.out.println("Uso: Fibonacci num");
7             System.out.println("Donde num = n° de términos");
8         }
9         else {
10            numTerm=Integer.valueOf(argumentos[0]);
11            System.out.print("1");
12            do {
13                aux=v2;
14                v2+=v1;
15                v1=aux;
16                System.out.print(", "+v2);
17            } while (++cont<numTerm);
18            System.out.println();
19        }
20    }
21 }
```

Listing 25. Ejemplo Fibonacci con ciclo *do-while*.

En este caso únicamente se muestra el primer término de la serie antes de iniciar el ciclo, ya que el segundo siempre se mostrará, porque el ciclo *do-while* siempre

se ejecuta una vez por lo menos.

Ciclo while

Sintaxis

```
while (<Expresión>)  
    <bloque instrucciones>
```

Al igual que en el ciclo *do-while* del apartado anterior, el bloque de instrucciones se ejecuta mientras se cumple una condición (mientras *Expresión* se evalúe verdadero), pero en este caso, la condición se comprueba antes de empezar a ejecutar por primera vez el ciclo, por lo que si *Expresión* se evalúa como falso en la primera iteración, entonces el bloque de instrucciones no se ejecutará ninguna vez.

Ejemplo:

```
1  //Fibonacci:  
2  public class Fibonacci3 {  
3      public static void main(String argumentos[]) {  
4          int numTerm,v1=1,v2=1,aux,cont=2;  
5          if (argumentos.length<1) {  
6              System.out.println("Uso: Fibonacci num");  
7              System.out.println("Donde num = n° de términos");  
8          }  
9          else {  
10             numTerm=Integer.valueOf(argumentos[0]);  
11             System.out.print("1,1");  
12             while (cont++<numTerm) {  
13                 aux=v2;  
14                 v2+=v1;  
15                 v1=aux;  
16                 System.out.print(", "+v2);  
17             }  
18             System.out.println();  
19         }  
20     }  
21 }
```

Listing 26. Ejemplo Fibonacci con ciclo *while*.

Como puede comprobarse, las tres construcciones de ciclo (*for*, *do-while* y *while*) pueden utilizarse indistintamente realizando unas pequeñas variaciones en el programa.

Salto

En Java existen dos formas de realizar un salto incondicional en el flujo normal de un programa: las instrucciones *break* y *continue*.

break La instrucción *break* sirve para abandonar una estructura de control, tanto de la alternativa (*switch*) como de las repetitivas o ciclos (*for*, *do-while* y *while*). En el momento que se ejecuta la instrucción *break*, el control del programa sale de la estructura en la que se encuentra.

Ejemplo:

```
1 public class Break {
2     public static void main(String argumentos[]) {
3         int i;
4         for (i=1; i<=4; i++) {
5             if (i==3)
6                 break;
7             System.out.println("Iteracion: "+i);
8         }
9     }
10 }
```

Listing 27. Ejemplo de uso de *break*.

Aunque el ciclo, en principio indica que se ejecute 4 veces, en la tercera iteración, *i* contiene el valor 3, se cumple la condición de *i==3* y por lo tanto se ejecuta el *break* y se sale del ciclo *for*.

continue La instrucción *continue* sirve para transferir el control del programa desde la instrucción *continue* directamente a la cabecera del ciclo (*for*, *do-while* o *while*) donde se encuentra.

Ejemplo:

```
1 public class Continue {
2     public static void main(String argumentos[]) {
3         int i;
4         for (i=1; i<=4; i++) {
5             if (i==3)
```

```

6         continue;
7         System.out.println("Iteración: "+i);
8     }
9 }
10

```

Listing 28. Ejemplo de uso de *continue*.

Puede comprobarse la diferencia con respecto al resultado del ejemplo del apartado anterior. En este caso no se abandona el ciclo, sino que se transfiere el control a la cabecera del ciclo donde se continúa con la siguiente iteración.

Tanto el salto *break* como en el salto *continue*, pueden ser evitados mediante distintas construcciones pero en ocasiones esto puede empeorar la legibilidad del código. De todas formas existen programadores que no aceptan este tipo de saltos y no los utilizan en ningún caso; la razón es que - se dice - que atenta contra las normas de las estructuras de control.

3.8.8 Arreglos

Para manejar colecciones de objetos del mismo tipo estructurados en una sola variable se utilizan los arreglos.

En Java, los arreglos son en realidad objetos y por lo tanto se puede llamar a sus métodos. Existen dos formas equivalentes de declarar arreglos en Java:

```
tipo nombreDelArreglo[ ];
```

o

```
tipo[ ] nombreDelArreglo;
```

Ejemplo:

```

1 int arreglo1[], arreglo2[], entero; //entero no es un arreglo
2 int[] otroArreglo;

```

También pueden utilizarse arreglos de más de una dimensión:

Ejemplo:

```

1 int matriz[][];
2 int [][] otraMatriz;

```

Los arreglos, al igual que las demás variables pueden ser inicializados en el momento de su declaración. En este caso, no es necesario especificar el número de elementos máximo reservado. Se reserva el espacio justo para almacenar los elementos añadidos en la declaración.

Ejemplo:

```
1 String Días[]={ "Lunes", "Martes", "Miércoles", "Jueves",  
2               "Viernes", "Sábado", "Domingo"};
```

Una simple declaración de un vector no reserva espacio en memoria, a excepción del caso anterior, en el que sus elementos obtienen la memoria necesaria para ser almacenados. Para reservar la memoria hay que llamar explícitamente a *new* de la siguiente forma:

```
new tipoElemento[ <numElementos> ];
```

Ejemplo:

```
1 int matriz[][];  
2 matriz = new int[4][7];
```

También se puede indicar el número de elementos durante su declaración:

Ejemplo:

```
int vector[] = new int[5];
```

Para hacer referencia a los elementos particulares del arreglo, se utiliza el identificador del arreglo junto con el índice del elemento entre corchetes. El índice del primer elemento es el cero y el del último, el número de elementos menos uno.

Ejemplo:

```
j = vector[0]; vector[4] = matriz[2][3];
```

El intento de acceder a un elemento fuera del rango del arreglo, a diferencia de lo que ocurre en C, provoca una excepción (error) que, de no ser manejado por el programa, será la máquina virtual quien aborte la operación.

Para obtener el número de elementos de un arreglo en tiempo de ejecución se accede al atributo de la clase llamado *length*. No olvidemos que los arreglos en Java son tratados como un objeto.

Ejemplo:

```
1 public class Array1 {
2     public static void main (String argumentos[]) {
3         String colores[] = {"Rojo", "Verde", "Azul",
4                             "Amarillo", "Negro"};
5
6         int i;
7         for (i=0; i<colores.length; i++)
8             System.out.println(colores[i]);
9     }
10 }
```

Listing 29. Ejemplo de uso de arreglo.

Usando al menos Java 5.0 (jdk 1.5) podemos simplificar el recorrido del arreglo:

```
1 public class Meses {
2
3     public static void main(String[] args) {
4         String meses[] =
5             {"Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
6             "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"};
7
8         //for(int i = 0; i < meses.length; i++ )
9         //    System.out.println("mes: " + meses[i]);
10
11        // sintaxis para recorrer el arreglo y asignar
12        // el siguiente elemento a la variable mes en cada ciclo
13        // instruccion "for each" a partir de version 5.0 (1.5 del jdk)
14        for(String mes: meses)
15            System.out.println("mes: " + mes);
16
17    }
18
19 }
```

Listing 30. Ejemplo de uso de arreglo 2.

3.8.9 Enumeraciones

Java desde la versión 5 incluye el manejo de enumeraciones. Las enumeraciones sirven para agrupar un conjunto de elementos dentro de un tipo definido. Antes, una manera simple de definir un conjunto de elementos como si fuera una enumeración era, por ejemplo:

```
1 public static final int TEMPO_PRIMAVERA = 0;
2 public static final int TEMPO_VERANO = 1;
3 public static final int TEMPO_OTONÑO = 2;
4 public static final int TEMPO_INVIERNO = 3;
```

Lo cual puede ser problemático pues no es realmente un tipo de dato, sino un conjunto de constantes enteras. Tampoco tienen un espacio de nombres definido por lo que tienen que definirse nombre. La impresión de estos datos, puesto que son enteros, despliega solo el valor numérico a menos que sea interpretado explícitamente por código adicional en el programa.

El manejo de enumeraciones en Java tiene la sintaxis de C, C++ y C# :

Sintaxis
<code>enum <nombreEnum> { <elem 1>, <elem 2>, ..., <elem n> }</code>

Por lo que para el código anterior, la enumeración sería:

```
enum Temporada { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
```

La sintaxis completa de enum es más compleja, ya que una enumeración en Java es realmente una clase, por lo que puede tener métodos en su definición. También es posible declarar la enumeración como pública, en cuyo caso debería ser declarada en su propio archivo.

Ejemplo:

```
1 enum Temporada { PRIMAVERA, VERANO, OTOÑO, INVIERNO }
2
3 public class EnumEj {
4
5
6     public static void main(String[] args) {
7         Temporada tem;
8         tem=Temporada.PRIMAVERA;
9         System.out.println("Temporada: " + tem);
10
11         System.out.println("\nListado de temporadas:");
12
13         for(Temporada t: Temporada.values())
14             System.out.println("Temporada: " + t);
15
16     }
17 }
```

Listing 31. Ejemplo de enumeración en Java.

3.8.10 Entrada desde consola en Java

La entrada tradicional en Java desde consola era evitada en libros y cursos en su etapa introductoria, esto debido a que era necesario incluir manejos de *Streams* o flujos, lo que comúnmente requiere mayor experiencia con el lenguaje y la programación orientada a objetos.

Sin embargo, a partir de la versión 1.5 del *jdk* se incluye la clase *Scanner* que proporciona comportamiento de lectura desde consola.

Ejemplo:

```
1  import java.util.Scanner;
2
3  public class ScannerTest {
4
5      public static void main(String[] args) {
6
7          String nom;
8          int edad;
9          Scanner in = new Scanner(System.in);
10
11         System.out.print("Nombre:");
12         // Lee una línea de consola
13         nom = in.nextLine();
14
15         System.out.print("Edad:");
16         // Lee un entero de consola
17         edad=in.nextInt();
18         in.close();
19
20         System.out.println("Nombre :"+nom);
21         System.out.println("Edad :"+edad);
22
23     }
24 }
```

Listing 32. Ejemplo de entrada de consola en Java con *Scanner*.

Operaciones similares a *nexInt()* y *nextLine()* existen para el resto de los tipos de datos.

La versión 1.6 del *jdk* incluye otra clase: *Console* la cual proporciona el comportamiento de lectura de una línea desde consola y lectura sin eco (tipo *password*) en la consola.

Ejemplo¹²:

```
1 import java.io.Console;
2
3 //no funciona en la consola de Eclipse
4 public class TestConsole {
5
6     public static void main(String... args ) {
7
8         // Obtener un objeto de consola
9         Console console = System.console();
10        if (console == null) {
11            System.err.println("No se obtuvo la consola.");
12            System.exit(1);
13        }
14
15        String usuario = console.readLine("Usuario:");
16
17        //Lee password y lo recibe en un arreglo de caracteres
18        char[] password = console.readPassword("Password: ");
19
20        if (usuario.equals("admin")
21            && String.valueOf(password).equals("secreto")) {
22            console.printf("Bienvenido %1$s.\n", usuario);
23
24        } else {
25            console.printf("Usuario o password inválido.\n");
26        }
27    }
28 }
```

Listing 33. Ejemplo de entrada de consola en Java con *Console*.

Como se pudo apreciar, esta clase también incluye operaciones de salida a consola. También simplifica la salida de caracteres especiales en la consola.

¹²Ejecutar directamente de consola ya que puede no ejecutarse correctamente en el algunos IDEs.

Argumentos de cantidad variable

Ahora, si fueron observadores sabrán que el último ejemplo nos trajo un nuevo tópico: el uso de los ... en la operación *main*. Este operador sirve para definir argumentos de cantidad variable, siendo el resultado almacenado en un arreglo del tipo especificado. **Podemos combinar los argumentos variables con otros argumentos, pero solo podemos meter un argumento variable y debe ir al final.**

Ejemplo:

```
1 public class TestArgs {
2     public static void main(String[] args) {
3         llamame1(new String[] {"a", "b", "c"});
4         llamame2("a", "b", "c");
5         // Otra opción:
6         // llamame2(new String[] {"a", "b", "c"});
7     }
8
9     public static void llamame1(String[] args) {
10        for (String s : args)
11            System.out.println(s);
12    }
13
14    public static void llamame2(String... args) {
15        for (String s : args)
16            System.out.println(s);
17    }
18 }
```

Listing 34. Ejemplo de entrada de argumentos de cantidad variable.

Otro ejemplo¹³, se presenta a continuación.

Ejemplo:

```
1
2 public class VarargsTest
3 {
4     // cálculo de promedio
5     public static double average( double... numbers )
6     {
7         double total = 0.0; // inicializar total
8     }
```

¹³Código original de [?]


```

9      for ( double d : numbers )
10         total += d;
11
12      return total / numbers.length;
13  }
14
15  public static void main( String args[] )
16  {
17      double d1 = 10.0;
18      double d2 = 20.0;
19      double d3 = 30.0;
20      double d4 = 40.0;
21
22      System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
23          d1, d2, d3, d4 );
24
25      System.out.printf( "Promedio de d1 y d2 es %.1f\n",
26          average( d1, d2 ) );
27      System.out.printf( "Promedio de d1, d2 y d3 es %.1f\n",
28          average( d1, d2, d3 ) );
29      System.out.printf( "Average de d1, d2, d3 y d4 es %.1f\n",
30          average( d1, d2, d3, d4 ) );
31  }
32  }

```

Listing 35. Ejemplo de entrada de argumentos de cantidad variable.

3.8.11 Paquetes

Las clases en Java son organizadas mediante paquetes. Un paquete es entonces el mecanismo para agrupar clases que están relacionadas, ya sea porque sirven a un propósito común o porque dependen unas de otras para realizar sus responsabilidades. Se usan los paquetes cuando importamos clases para ser incorporadas en nuestro código, pero si queremos especificar el paquete al que pertenecen nuestras clases podemos hacerlo.

Sintaxis
<code>package [<ruta>]<nombre paquete></code>

Si no se define el nombre del paquete, por omisión se considera el nombre del directorio donde la clase se encuentra definida.

Capítulo 4

Introducción a Ruby

4.1 Introducción

Ruby es un lenguaje definido para ser dinámico, reflexivo y orientado a objetos. Combina una sintaxis inspirada en Perl y similar a las características orientadas a objetos de Smalltalk. También comparte ciertas características con Python, Lisp, Dylan, y CLU.

Ruby fue creado por **Yukihiro Matsumoto** con la idea de crear un lenguaje que balanceara la programación funcional con la programación imperativa.

El lenguaje fue liberado desde un inicio como *open source* (1995) y en los últimos años ha crecido su aceptación masivamente.

Matsumoto dice haber puesto énfasis en crear un lenguaje productivo y divertido, siguiendo los principios de buen diseño de interfaz con el usuario. Remarca que el diseño de sistemas necesita –también– enfatizar las necesidades humanas, en lugar de las de la computadora.

Ruby es actualmente un lenguaje interpretado aunque se pretende que en futuras versiones sea semi-compilado y ejecutado por una máquina virtual, de manera similar a Java. Además existe un número amplio de implementaciones de Ruby con diferentes aproximaciones.



Ruby's architect,
Yukihiro Matsumoto

4.2 Características

Ruby es principalmente un lenguaje orientado a objetos, pero también es descrito como un lenguaje multiparadigma: permite programación procedural, con orientación a objetos y declaraciones funcionales.

Un resumen de sus características puede verse enseguida:

- Orientado a objetos
- 4 niveles de alcance de variables: global, clase, instancia y local

- Manejo de excepciones
- Expresiones regulares nativas al nivel del lenguaje (*perl-like*)
- Sobrecarga de operadores
- Recolector automático de basura
- Biblioteca dinámica compartida en la mayoría de las plataformas
- Soporta introspección, reflexión y metaprogramación
- Soporta *continuations* y *generators*

4.2.1 Comparado con C

Similitudes

- Puedes programar proceduralmente si lo deseas, pero aún será orientado a objetos internamente.
- La mayoría de los operadores son los mismos. Pero no cuenta con ++ o –.
- Se pueden tener constantes, aunque no hay una instrucción *const*.
- Las cadenas van entre comillas y son mutables.
- Se cuenta con un depurador en línea.

Diferencias

- Los objetos tienen un tipo fuerte y las variables no tienen tipo.
- No cuenta con macros o preprocesador.
- No tiene enmascaramiento.
- No tiene apuntadores, ni aritmética de apuntadores.
- No tiene *typedef*, *sizeof*, ni enumeraciones.
- No archivos de encabezados.
- Es interpretado en tiempo de ejecución, por lo que no hay código compilado o *bytecode* de ningún tipo (hasta la versión 1.8).
- Cuenta con recolector de basura.
- Los argumentos son pasados por referencia, no por valor.

- No usa ';' obligatoriamente para finalizar instrucciones
- Condiciones para *if* y *while* van sin paréntesis.
- Paréntesis para llamadas a métodos son frecuentemente opcionales.
- Usualmente no se usan llaves, se finalizan las instrucciones de múltiples líneas con una palabra clave.
- No hay declaraciones de variables. Asignas nombre conforme los necesitas.
- Sólo falso y nulo evalúan como falso. Cualquier otro valor es verdadero (incluyendo 0 –cero-)
- No hay tipo *char*.
- Cadenas no terminan con un valor nulo.
- Los arreglos automáticamente se agrandan conforme vas necesitando más elementos.

4.2.2 Comparado con C++

Similitudes

- Public, protected y private realizan actividades similares.
- Puedes poner tu código en módulos, similar a espacios de nombre en C++.
- Excepciones trabajan de forma similar.

Diferencias

- No hay referencias explícitas. En Ruby cada variable es un nombre automáticamente desreferenciado para un objeto.
- El constructor es llamado initialize en lugar de usar el nombre de la clase.
- Todos los métodos son siempre virtuales.
- Nombres de atributos de clase siempre empiezan con @@.
- No es posible acceder directamente variables miembros. Todos los atributos deben ser accedidos a mediante métodos.
- Se usa self en lugar de *this*.
- Algunos métodos terminan con '?' o '!'. Es parte del nombre del método.

- No hay herencia múltiple.
- Existen algunas reglas con nombres (e.g., nombre de clases empiezan con mayúscula, variables inician con minúscula.)
- Solo dos tipos de clases contenedoras: *Array* y *Hash*.
- No hay conversiones de tipos automáticas.
- Multihilos son implementados en el intérprete (*green threads*). No son hilos nativos.
- Existe una biblioteca para pruebas de unidad como parte estándar del lenguaje.

4.2.3 Comparado con Java

Similitudes

- La memoria es manejada automáticamente mediante un recolector de basura.
- Los objetos son fuertemente tipados.
- Hay métodos públicos, privados y protegidos.
- Tiene herramientas de documentación embebidas (la de Ruby se llama RDoc). La documentación generada por rdoc se ve muy similar a la generada por javadoc.

Diferencias

- No necesitas compilar tu código fuente. Directamente lo ejecutas.
- Hay distintos conjuntos de herramientas para interfaz gráfica.
- Se usa la palabra clave *end* después de definir clases, en vez de tener que poner llaves encerrando el código.
- Tienes *require* en vez de *import*.
- Todas las variables de instancia son privadas. Desde afuera, todo se accede usando métodos.
- Los paréntesis en las llamadas a los métodos usualmente son opcionales y a menudo son omitidos.
- Todo es un objeto, incluyendo los números como 2 y 3,14159.

- No hay validación estática de tipos de datos.
- Los nombres de variables son sólo etiquetas. No tienen un tipo de dato asociado.
- No hay declaración de tipo de datos. Simplemente se asigna a nuevos nombres de variable a medida que se necesita (por ejemplo $a = [1, 2, 3]$ en vez de `int[] a = 1, 2, 3;`).
- No hay transformación de tipos (*casting*). Simplemente se llama a los métodos. Tus pruebas unitarias deberían avisarte antes de que ejecutes el código si habrá una excepción.
- Es `foo = Foo.new("hola")` en vez de `Foo foo = new Foo("hola")`.
- El constructor siempre se llama "initialize" en vez del nombre de la clase.
- Tienes *mixins* en vez de interfaces.
- Es *nil* en vez de *null*.

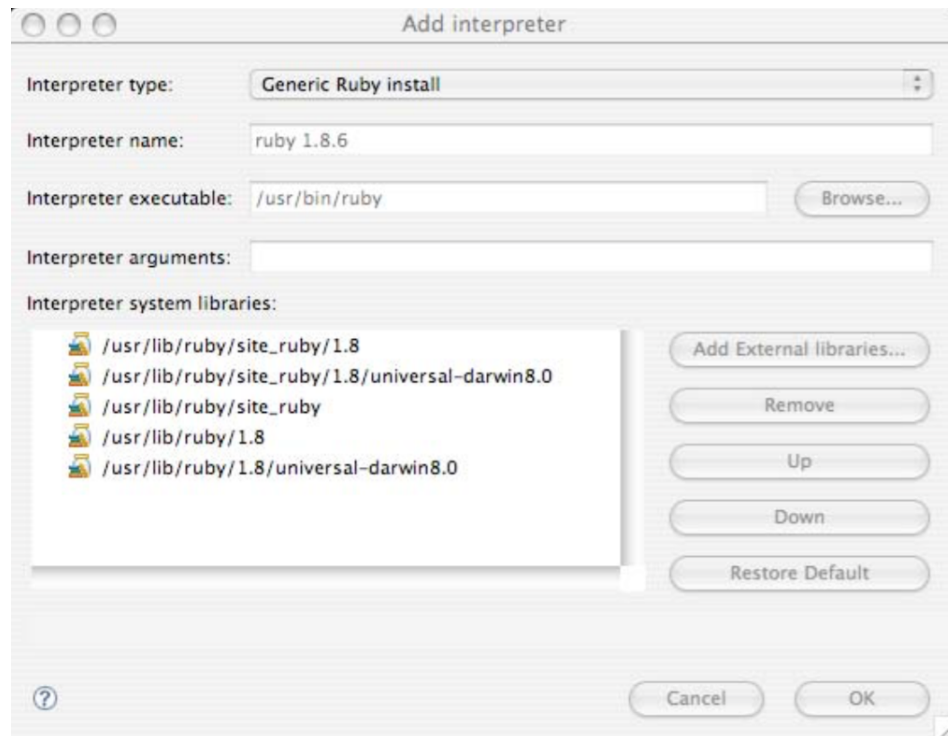
4.3 Herramientas

Existen dos herramientas básicas en Ruby:

- **ruby**. Es el intérprete del lenguaje. Puede recibir expresiones del lenguaje como parámetros o archivos con programas.
- **irb** (o **fxri** en algunas versiones para Windows). Este es Ruby interactivo (*Interactive RuBy*) que permite recibir expresiones del lenguaje e ir las interpretando línea por línea, como cualquier lenguaje interpretado. Además:
- **ri**. Documentación de clases estándar de ruby.

También es posible integrar al intérprete al IDE de Eclipse. Para esto se debe agregar el plugin llamado *Ruby Development Tools* (RDT) aparte de tener instalado el intérprete en la computadora. El plugin puede ser encontrado en: <http://rubyclipse.sourceforge.net/>. Tiene que agregarse, como cualquier otro plugin en la herramienta, mediante la opción de actualización del software en el menú de ayuda de Eclipse.

El plugin debe configurarse indicando la ubicación del intérprete:



4.4 Ruby: Fundamentos del lenguaje

4.4.1 Convenciones léxicas

Espacios en blanco

Mientras una expresión del tipo $a + b$ es interpretada como $a + b$, donde a es una variable. El resultado puede ser diferente en casos ambiguos. Por ejemplo, si a es el nombre de una función, entonces una expresión:

$a + b$

es interpretada como:

$a(+b)$

Final de instrucciones

Ruby interpreta `;` y el espacio en blanco como el final de una instrucción. Debido a esto, Ruby interpreta los símbolos `+`, `-` y `\` como continuación de una instrucción.

Comentarios

Comentarios en Ruby son representados con `#` :

```
# Este es un comentario
```

Comentarios de más de una línea usan `= begin` y `= end`, los cuales deben estar al comienzo de una línea:

```
1 =begin
2 Este es un comentario
3 =end
```

Identificadores

Cualquier nombre de constante, variables y métodos usado como identificador es distinguido por Ruby si usa minúsculas o mayúsculas.

4.4.2 Literales

Enteros

Los números enteros son instancias de la clase *Fixnum* o *Bignum*.

```
1 123                # decimal
2 0377              # octal
3 0xff              # hexadecimal
4 0b1011            # binary
5 ?a                # código para 'a'
6 12345678901234567890 # Bignum: entero de longitud infinita
```

Flotantes

Los números de punto flotante son instancias de la clase *Float*.

```
1 123.4
2 1.0e6                # notación científica
3 4e+20                # exponencial
```

Cadenas

Una cadena es un arreglo de bytes y una instancia de la clase *String*:

```
1 "una cadena"      # permite sustitución y notación con \
2 'otra cadena'     # no permite sustitución y solo \ \ ó \ '.
```

Concatenación

Cadenas adyacentes son concatenadas :

```
'una' 'cadena' # es igual a 'una cadena'
```

4.4.3 Variables

En Ruby existen 5 tipos de variables, usando caracteres especiales para diferenciar entre los distintos tipos de variables, lo que ayuda a identificar el tipo de variable visualmente:

- Variable global
- Variable de instancia
- Variable de clase
- Variable local
- Constante

y además:

- Pseudo-variable

Variable global

Visibles a través de todo el programa y deben iniciar con el símbolo \$, por ejemplo:

```
$soyGlobal
```

Una variable global no inicializada tiene el valor de *nil*. Existen además variables globales psobreescribas que contienen información sobre el programa en ejecución.

Variable de instancia

Las variables de instancia pertenecen a un objeto y son lo que también es conocido en objetos como atributos. Estas son visibles dentro de un objeto en particular y deben comenzar con @, por ejemplo:

```
@soyVariableDeInstancia
```

Al igual que las variables globales, estas tienen el valor de *nil* si no han sido inicializadas.

Variable de clase

Las variables de clase son visibles, como su nombre lo dice, en la clase y para todos los objetos de la misma. Comienzan con `@@`, por ejemplo:

```
@@variableDeClase
```

Estas variables deben ser inicializadas antes de que puedan ser usadas en los métodos. El uso de una variable de clase no inicializada produce un error. Además, estas clases son compartidas por descendientes de las clases donde fueron definidas.

Variable local

Son válidas dentro del ámbito local definido y deben empezar con una minúscula o con el símbolo `_`. El ámbito puede ser el que defina una clase, módulo, definición, `do`–`end`.

Constante

Deben empezar con una letra mayúscula y pueden ser definidas dentro de una clase o módulo y serán visibles dentro de ese ámbito. Una constante definida fuera de un clase o módulo será vista globalmente. Es posible reasignar un valor a una constante, pero esto producirá una advertencia (pero no un error).

Pseudo-variable

Pseudo-variables tienen la apariencia de variables locales pero su comportamiento es el de constantes. Ejemplo de estas variables son *self*, *true*, *false*, *nil*.

4.4.4 Operadores

Asignación

La asignación funciona con el operador `=`. Asignar variables locales también sirve como declaración de la variable. La variable existe hasta el final del alcance donde la variable es declarada.

También se cuenta con asignación abreviada como en los lenguajes C/C++ y Java:

```
1 += -= *= /= %= **= <<= >>= &= |= ^= &&= ||=
```

Asignación paralela

```
destino[, destino...][, *destino] = expr[, expr...][, *expr]
```

Identificadores destino reciben la asignación de la correspondiente expresión en el lado derecho. Si el último destino (lado izquierdo) tiene como prefijo un ***, el resto de los valores en el lado derecho se asigna en ese destino como un arreglo. Si el *** esta en el último elemento del lado derecho, el conjunto de elementos son expandidos antes de su asignación.

Operadores lógicos

&& o *and*. Regresa *true* si ambos operandos son verdaderos. Si el operando izquierdo es falso, regresa ese valor, en caso contrario regresa el valor del operando derecho.

|| o *or*. Regresa *true* si cualquiera de los operandos es verdadero. Si el valor del operando izquierdo es *true*, regresa el valor de ese operando, de otro modo regresa el valor del operando derecho.

Un aspecto interesante aquí es que los operadores *and* y *or* tienen una precedencia muy baja, de hecho tienen la menor de las precedencias entre los operadores.

4.4.5 Operador ternario

El operador ternario *? :* es el operador condicional similar al de C/C++ y Java.

```
a? b : c
```

4.4.6 Operador *defined?*

Este es un operador que puede determinar si una expresión esta definida. Regresa una descripción de la expresión, o nulo si la expresión no esta definida.

```
defined? variable
```

Por ejemplo:

```
1 defined? a
2 defined? $_
```

Puede ser usado para verificar una llamada a un método, opcionalmente incluyendo sus argumentos.

Prioridad de operadores

A continuación se presenta los operadores más comunes de Ruby en orden de precedencia, de mayor a menor:

```
1  ::
2  []
3  **
4  +(unario) -(unario)
5  * / %
6  + -
7  << >>
8  &
9  | ^
10 > >= < <=
11 <=> == !=
12 &&
13 ||
14 .. ...    # Operadores de rango (... excluyendo el límite derecho)
15 ?:
16 =        # (y operadores abreviados como +=, -=, etc.)
17 not
18 and or
```

Operadores que no pueden ser sobreescritos

Los siguientes operadores no pueden ser sobreescritos (no son métodos):

```
1  ...
2  !
3  not
4  &&
5  and
6  ||
7  or
8  ::
9  =
10 +=, -=, # (y el resto de las asignaciones abreviadas)
11 ? :
```

4.4.7 Arreglos

Un arreglo en Ruby es una clase contenedora que contiene una colección de objetos. **Cualquier** tipo de objetos pueden ser almacenados en un arreglo, inclusive pudiendo contener elementos de distinto tipo en un mismo arreglo. Otra característica es que el arreglo aumenta de tamaño conforme se añaden elementos. Un arreglo es representado con sus elementos entre corchetes [] :

```
[ ]           #Arreglo vacío
[1, 2, 3]     #Arreglo de 3 elementos
```

Los arreglos pueden ser asignados:

```
ar1= []
ar2= [1, 2, 3]     # => [1, 2, 3]
```

Un arreglo puede añadir fácilmente un elemento mediante el operador `<<`. Ruby dinámicamente ajusta el tamaño del arreglo al añadir o remover los elementos:

```
ar3= ar2 << "otro"   # => [1, 2, 3, "otro"]
```

El operador `<<` modifica el operando izquierdo, por lo que la modificación de un arreglo puede hacerse directamente como:

```
ar3<<"otro mas"     # => [1, 2, 3, "otro", "otro mas"]
```

De hecho, en la penúltima expresión `ar3` en realidad está recibiendo la referencia de `ar2` al cual se le añadió la cadena "otro".

También es posible asignar a una posición fuera del arreglo actual. Por ejemplo:

```
ma= [3,4,5]         # => [3, 4, 5]
```

Y luego:

```
ma[3]=10            # => [3, 4, 5, 10]
```

E inclusive:

```
1 ma[11]="Al infinito y mas alla"    # => [3, 4, 5, 10, nil, nil, nil,
2                                   # nil, nil, nil, nil, "Al infinito y mas allá"]
```

Como en C++ y Java, en Ruby el índice de un arreglo comienza con cero.

```
1 a = [1, 2, 3, [4, 5, 6]]    # => [1, 2, 3, [4, 5, 6]]
2
3 a[0]                        # => 1
```

El método *size* puede ser utilizado para conocer el número de elementos del arreglo:

```
1 a.size                      # => 4
2 a[3]                        # => [4, 5, 6]
3 a[3].size                   # => 3
4 a[3][0]                     # => 4
```

El tamaño del arreglo es validado:

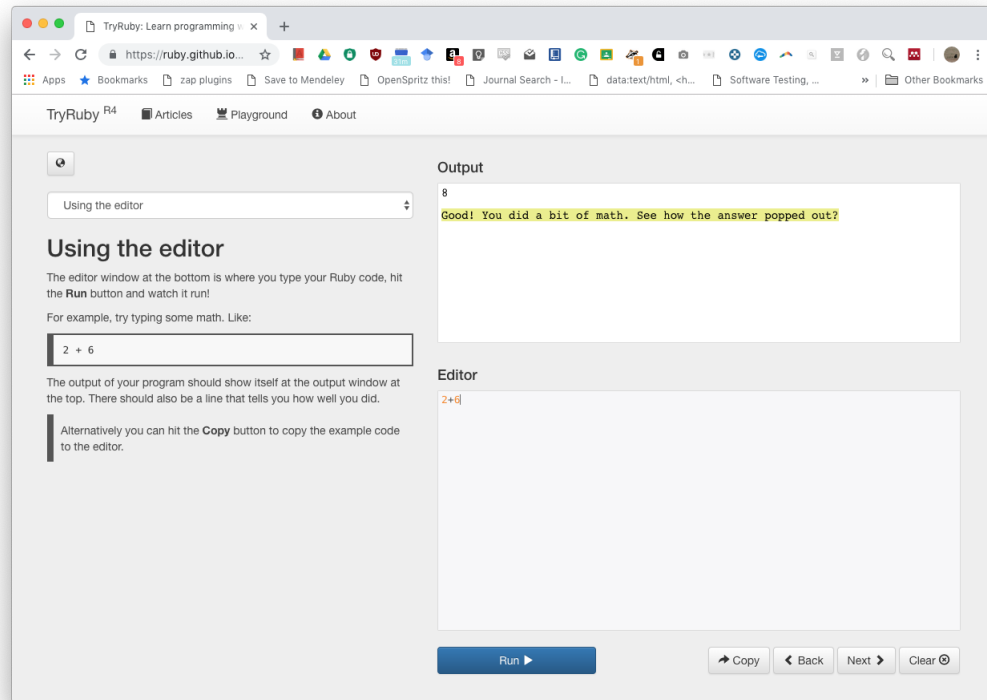
```
a[5]                          # => nil
```

Es posible hacer uso de valores negativos como índices, y estos se tomarán de la última posición (-1) hasta la posición negativa del tamaño del arreglo (también posición 0:

```
1 a[-1]                       # => [4, 5, 6]
2 a[a.size*-1]                # => 1
```

4.4.8 Probando Ruby

¿Porque no empezar a probar ruby siguiendo el tutorial el línea? Éste se encuentra disponible en : <https://ruby.github.io/TryRuby/>



4.4.9 Estructuras de control

Condicional if

Se ejecuta si la condición es verdadera.

Sintaxis

```
if condicional [then]
  instrucciones
[elsif condicional [then]
  instrucciones]...
[else
  instrucciones]
end
```

El *if* puede ser usado como un modificador de una declaración:

```
code if condicional
```

Ejemplos:

```
1 if x < 5 then
2   declaracion1
3 end
4
5 if x < 5 then
6   declaracion1
7 else
8   declaracion2
9 end
10
11 declaracion1 if y == 3
12
13 x = if a>0 then b else c end
```

Condicional *unless*

Ejecuta código si la condición es falsa, en caso contrario ejecuta otro bloque de instrucciones.

Sintaxis

```
unless condicional [then]  
  instrucciones  
[else
```

El *unless* puede ser usado como un modificador de una declaración:

```
instrucciones unless condicional
```

Ejemplos:

```
1 unless x >= 5 then  
2   declaracion1  
3 end  
4  
5 unless x < 5 then  
6   declaracion1  
7 else  
8   declaracion2  
9 end  
10  
11 declaracion1 unless y != 3  
12  
13 x = unless a<=0 then c else b end
```

Case

Compara la expresión especificada en *case* con la expresión especificada en *when* y ejecuta el código correspondiente. La cláusula *else* se ejecuta en el caso de que ningún segmento *when* sea ejecutado.

Sintaxis

```
case expresión
[when expresión[, expresión...] [then]
  instrucciones]...
[else
  instrucciones]
end
```

Ejemplo¹:

```
1 case cad
2   when "algun valor"
3     puts "opcion 1"
4   when "otro valor"
5     puts "opcion 2"
6   when /char/
7     puts "opcion 3"
8   else
9     puts "opcion 4"
10 end
```

Ciclo *while*

Se ejecuta el conjunto de instrucciones mientras la condición es verdadera. La condición puede ser separada del conjunto de instrucciones mediante la palabra reservada *do*, una línea nueva, el símbolo ' \ ' , o un ' ; '.

¹Recordar que el tipo *char* no existe, aquí / / indica una expresión regular que en este caso solo contiene el patrón *char*. Finalmente, es un objeto de tipo *Regexp*.

Sintaxis

```
while condicional [do]  
  instrucciones  
end
```

El *while* puede ser usado como un modificador de una declaración:

```
instrucciones while condicional
```

O:

```
begin  
  instrucciones  
end while condicional
```

Ejecuta instrucciones mientras la condición es verdadera. En el caso del *while*, entre las cláusulas *begin* y *end*, éste se ejecuta una vez **antes** de evaluar la condición.

Ciclo *until*

El ciclo *until* se ejecuta el conjunto de instrucciones mientras la condición es falsa (hasta que la condición se cumpla). Puede ser separada del código por la palabra reservada *do*, un salto de línea o un *;'*. De igual forma que el *while*, *until* puede ser usado como modificador de una declaración.

Sintaxis
<pre>until condicional [do] instrucciones end</pre>
<p>O:</p> <pre>instrucciones until condicional</pre> <p>O:</p>
<pre>begin instrucciones end until condicional</pre>

Ciclo *for*

Ejecuta el conjunto de instrucciones por cada elemento en la expresión. La expresión en el `for` puede ir separada por la palabra reservada `do`, un salto de línea, o un `;/`.

Sintaxis
<pre>for variable in expresion [do] instrucciones end</pre>

Instrucciones `break`, `next`, `redo`, `retry`

break Termina un ciclo *while* o *until*. También finaliza un método con un bloque asociado si es usado dentro del bloque, con el método regresando el valor de nulo.

next Salta al punto en que se evalúa la condición de un ciclo. También termina la ejecución de un bloque si es llamado dentro de éste.

redo Salta al punto inmediatamente posterior a la evaluación del ciclo.

```
1 for i in 0..5
2   if i < 2 then
3     puts "El valor de la variable es #{i}"
4     redo
5   end
6 end
```

Listing 36. Ejemplo *redo* en Ruby.

retry En una iteración se reinicia la llamada a la iteración. Los argumentos son reevaluados.

```
1
2 for i in 1..5
3   retry if i > 2
4   puts "El valor de la variable es #{i}"
5 end
```

Listing 37. Ejemplo *retry* en Ruby.

Instrucciones *BEGIN* y *END*

BEGIN Permite declarar un conjunto de instrucciones a ejecutarse antes de que el programa se ejecute.

Sintaxis

```
BEGIN {
  instrucciones
}
```

END Permite declarar un conjunto de instrucciones a ejecutarse antes de finalizar la ejecución del interprete.

Sintaxis

```
END {  
  instrucciones  
}
```

Algunos ejemplos:

```
1  
2 # Ciclo 1 (while)  
3 i=0  
4 while i < list.size do  
5   print "#{list[i]} "  
6   i += 1  
7 end  
8  
9 # Ciclo 2 (until)  
10 i=0  
11 until i == list.size do  
12   print "#{list[i]} "  
13   i += 1  
14 end  
15  
16 # Ciclo 3 (for)  
17 for x in list do  
18   print "#{x} "  
19 end  
20  
21 # Ciclo 4 (loop)  
22 i=0  
23 n=list.size-1  
24 loop do  
25   print "#{list[i]} "  
26   i += 1  
27   break if i > n
```

```
28 end
29
30 # Ciclo 6 (loop)
31 i=0
32 n=list.size-1
33 loop do
34   print "#{list[i]} "
35   i += 1
36   break unless i <= n
37 end
38
39 # Ciclo 7 (for)
40 n=list.size-1
41 for i in 0..n do
42   print "#{list[i]} "
43 end
44
45 10.times do
46   puts "hello"
47 end
48 Programa 37
```

Listing 38. Ejemplos ciclos en Ruby.

```
1
2 car = "Patriot"
3
4 manufacturer = case car
5   when "Focus" then "Ford"
6   when "Navigator"
7     "Lincoln"
8   when "Camry"
9     "Toyota"
10  when "Civic"
11    "Honda"
12  when "Patriot" then "Jeep"
13  when "Jetta" then "VW"
14  when "Ceyene" then "Porsche"
15  when "Outback"
16    "Subaru"
17  when "520i": "BMW"
```



```
18   when "Tundra": "Nissan"
19   else "Desconocido"
20 end
21
22 puts "El " + car + " es fabricado por " + manufacturer
```

Listing 39. Ejemplo de *case* en Ruby.

```
1
2 calif = 70
3 result = case calif
4 when 0..59 then "Reprobado"
5 when 61..70 then "Aprobado... apenas"
6 when 71..80
7     "Aprobado"
8 when 81..100
9     "Excelente"
10 else "Resultado invalido"
11 end
12 puts result
```

Listing 40. Ejemplo 2 de *case* en Ruby.

```
1
2 for j in 1..5 do
3     for i in 1..5 do
4         print i, " "
5     end
6 puts
7 end
```

Listing 41. Ejemplo de *for* en Ruby.

```
1 for i in 1..8 do
2     puts i
3 end
4
```

Listing 42. Ejemplo 2 de *for* en Ruby.

4.4.10 Entrada y Salida básica en Ruby

Ruby proporciona instrucciones básicas de entrada y salida. Para Desplegar en la consola, las instrucciones básicas son *puts*, *print* y *printf*:

- *puts*. Despliega en la consola y añade un enter al final.
- *print*. Despliega en la consola pero no añade el enter o salto de línea al final.
- *printf*. Permite formatear la salida de variables de forma similar a C y Java 5.

Ejemplo:

```
1 puts "puts funciona"
2 puts " con saltos de linea."
3
4 print "print funciona"
5 print " sin saltos de linea."
6
7 printf("\n\nprintf formatea números como %7.2f, y cadenas como %s.", 3.14156, "esta")
```

Listing 43. Ejemplo de salida en Ruby.

La manera más simple de leer una cadena en Ruby es ocupando la función *gets*:

```
1 print "Introduce tu nombre: "
2 nom= gets
3
```

Listing 44. Ejemplo de entrada simple en Ruby.

Capítulo 5

Introducción a Python

5.1 Introducción

Python es un lenguaje dinámico y con características de orientado a objetos que es muy popular para desarrollo en Web, aunque cuenta también con características de programación funcional. Es similar a lenguajes como Ruby, Perl y Scheme pero también tiene influencias de lenguajes como Java y C.

Fue desarrollado en 1990 por *Guido van Rossum* y es un lenguaje que se ejecuta en las principales plataformas de hardware y sistemas operativos. Actualmente, junto con Ruby, Python es uno de los lenguajes orientados a objetos más usados para desarrollo de web dinámico ¹.

Existen tres principales implementaciones de Python:

- *Python / Cpython*. También llamada solamente Python, debido a que es la implementación más popular. La razón es que es la que tiene un desarrollo más completo, actualizado y de rápida ejecución².
- *Jython*. Es una implementación de Python para ejecutarse en máquinas virtuales de Java (JVM), de manera similar a Scala. Puede hacer uso de la biblioteca de clases de Java.
- *IronPython*. Es una implementación de Python para la CLR (*Common Language Runtime*) de Microsoft (.NET). Puede usar las bibliotecas de clases de .NET

¹Un artículo interesante de despedida a Guido por Dropbox donde mencionan su trabajo en Python [en el blog de Dropbox](#)

²Ver: python.org

5.2 Herramientas

El principal programa para usar Python lleva precisamente este nombre. *python* es al mismo tiempo el intérprete y el compilador del lenguaje. El programa genera código de bytes que es almacenado en programas *.pyc* o *.pyo*. Estos archivos son generados automáticamente cuando el archivo fuente es actualizado.

Python puede ejecutar código de 2 formas:

1. Interactivamente. Se ejecuta *python* desde el *prompt* de la consola:

```
> python
Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

2. Ejecución interpretada de archivos. *python* seguido del nombre del *script* a ejecutar.

```
> python programa.py
```

Además, Python incluye un sencillo ambiente de desarrollo llamado IDLE (*Integrated DeveLopment Environment*), el cual ofrece un *shell* similar al intérprete de Python con ligeras funcionalidades añadidas; además de incluir un editor de texto, visores y un depurador interactivo. Python puede ser usado desde otros IDEs, tales como Eclipse y NetBeans.

5.3 Fundamentos de Python

5.3.1 Convenciones léxicas

Un programa en Python está formado por una secuencia de líneas lógicas que pueden estar formadas por una o más líneas físicas. Una línea no lleva un delimitador como en otros lenguajes. En cambio, si la línea es muy larga, dos líneas físicas puede unirse con una diagonal '**'. Aunque Python automáticamente une dos líneas físicas si un paréntesis, corchete o llave no ha sido cerrado.

La **indentación** es importante para Python. A diferencia de muchos lenguajes, Python no usa llaves u otros medios (como *begin-end*) para delimitar bloques de instrucciones. La indentación es la forma en que los bloques son delimitados en Python.

5.3.2 Literales

Python tiene tipos definidos para tipos de datos básicos. Estos son objetos que también pueden ser usados como literales.

Por ejemplo, las literales enteras pueden ser escritas en decimal:

```
>>> 123
123
```

o hexadecimal:

```
>>> 0x17
23
```

Números flotantes se escriben con un punto y tienen el equivalente a un *double* en C.

5.3.3 Variables

Una variable es un espacio para almacenar datos modificables, en la memoria de una computadora, en Python una variable se define por la sintaxis:

```
nombre_de_la_variable= valor_de_la_variable
```

Cada variable tiene un nombre y un valor, el cual define a la vez el tipo de datos de la variable. Para nombrarlos es necesario un nombre descriptivo y en minúsculas. Pueden utilizarse nombres compuestos pero las palabras se separan por guiones bajos.

Existen otros tipos de datos que no requieren ser modificados a lo largo del programa, y son llamadas constantes. Los nombres de las constantes deben estar escritos con mayúsculas, y se deben separar las palabras por guiones bajos al igual que los nombres de las variables. Para imprimir un valor de pantalla, en Python, se utiliza la palabra clave *print*:

```
1 mi_variable = 15
2 print (mi_variable)
```

Esto imprimirá el valor de la variable *mi_variable* en la pantalla.

Cuando una variable tiene el valor de nulo, se usa la palabra reservada *None*.

Símbolo	Significado	Ejemplo	Resultado
+	Suma	10+5	15
-	Resta	12-7	5
-	Negación	-5	-5
*	Multiplicación	7*5	35
**	Exponente	2**3	8
/	División	12.5/2	6.25
//	División entera	12.5//2	6.0
%	Módulo	27%4	3

Cuadro 5.1. Operadores aritméticos en Python

Símbolo	Significado	Ejemplo	Resultado
==	Igualdad	5==7	Falso
!=	Diferencia	4 != 5	Verdadero
<	Menor que	5 < 7	Verdadero
>	Mayor que	4 > 8	Falso
<=	Menor o igual que	5 <= 5	Verdadero
>=	Mayor o igual que	4 >= 5	Falso

Cuadro 5.2. Operadores relacionales en Python

5.3.4 Operadores

Operadores aritméticos

Los operadores aritméticos que son utilizados en Python son mostrados en el Cuadro 5.1

El siguiente es un ejemplo donde se utilizan los operadores aritméticos:

```

1 >>> monto_bruto = 175
2 >>> tasa_interes = 12
3 >>> monto_interes = monto_bruto * tasa_interes / 100
4 >>> tasa_bonificacion = 5
5 >>> importe_bonificacion = monto_bruto * tasa_bonificacion / 100
6 >>> monto_netto = (monto_bruto - importe_bonificacion) + monto_interes
7 >>> monto_netto
8 187.25

```

Operadores relacionales

Ver Cuadro 5.2 con el listado de los operadores relacionales.

Símbolo	Ejemplo	Resultado
and	5==7 and 7<12	Falso
or	7>5 or 9<12	Verdadero
xor (o excluyente)	4==4 xor 9>3	Falso

Cuadro 5.3. Operadores lógicos en Python

Curiosamente, mientras muchos lenguajes únicamente permiten usar a los operadores relacionales para comparar pares de elementos. En Python permiten formar expresiones con elementos adicionales.

```

1 >>> 5>3>10
2 False
3 >>> 5>10>1
4 False
5 >>> 5>3>2
6 True
7 >>> 5>2>3
8 False
9 >>> 5>2<3
10 True

```

Operadores lógicos

Y para evaluar más de una condición simultáneamente se utilizan los operadores lógicos presentados en el Cuadro 5.3

5.3.5 Tipos de datos

Una variable puede contener valores de diversos tipos. Entre ellos:

```

1 Cadena de texto (String)
2 mi_cadena = "Hola mundo"
3 Número entero
4 edad = 35
5 Número hexadecimal
6 edad = 0x23
7 Número real
8 precio = 7435.28
9 Booleano (verdadero / falso)

```

```
10 verdadero = True
11          falso = False
```

Estos son algunos tipos de datos sencillos, además en Python existen otros tipos de datos complejos que admiten una colección de datos, como las **tuplas**, las **listas** y los **diccionarios**.

Listas

Python no tiene el concepto de arreglos. La secuencia de elementos más usada en el lenguaje son las listas. Las listas son colecciones de elementos de tipos arbitrarios y sin un tamaño fijo[?].

Son similares a los arreglos en otros lenguajes, con la diferencia de que son de tamaño dinámico y pueden contener elementos de distinto tipo.

```
mi_lista = ['cadena de texto', 15, 2.8, 'otro dato', 25]
```

A los datos de las listas se accede mediante el índice entre corchetes. Sus datos son mutables.

```
1 mi_lista[2] = 3.5
2 print (mi_lista)  # Devuelve ['cadena de texto', 15, 3.5, 'otro dato', 25]
```

También pueden agregarse nuevos datos a la lista,

```
1 mi_lista.append('Nuevo dato')
2 print (mi_lista)
3      #Devuelve ['cadena de texto', 15, 3.5, 'otro dato', 25, 'Nuevo dato']
```

Ejemplos:

```
1 >>> lista=[123, 'xxx', 3.14]
2 >>> len(lista)
3 3
4 >>> lista[0]
5 123
6 >>> lista + [4, 5, 6]
7 [123, 'xxx', 3.14, 4, 5, 6]
8 >>> lista * 2
9 [123, 'xxx', 3.14, 123, 'xxx', 3.14]
```



```
10 >>> lista
11 [123, 'xxx', 3.14]
12
13 >>> lista=lista+[4,5,6]
14 >>> lista
15 [123, 'xxx', 3.14, 4, 5, 6]
16 >>> lista.append('zzz')
17 >>> lista
18 [123, 'xxx', 3.14, 4, 5, 6, 'zzz']
19 >>> lista.pop(2)
20 3.14
21 >>> lista
22 [123, 'xxx', 4, 5, 6, 'zzz']
23 >>> orden=['c', 'a', 'b']
24 >>> orden.sort()
25 >>> orden
26 ['a', 'b', 'c']
27 >>> orden.reverse()
28 >>> orden
29 ['c', 'b', 'a']
30 >>> lista
31 [123, 'xxx', 4, 5, 6, 'zzz']
32 >>> lista[100]
33 Traceback (most recent call last):
34   File "<stdin>", line 1, in <module>
35 IndexError: list index out of range
36 >>> lista[100]=1
37 Traceback (most recent call last):
38   File "<stdin>", line 1, in <module>
39 IndexError: list assignment index out of range
40
41
42 >>> matriz=[[1, 2, 3],
43 ... [4, 5, 6],
44 ... [7, 8, 9]]
45 >>> matriz
46 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
47 >>> matriz[1]
48 [4, 5, 6]
49 >>> matriz[1][2]
50 6
```

Tuplas

Una tupla es una variable que permite almacenar varios datos inmutables de tipos diferentes.

```
mi_tupla = ('Cadena de texto', 15 , 2.8 , 'otro dato', 25)
```

Se puede acceder a cada uno de estos datos mediante su índice correspondiente, siendo el 0 el índice del primer elemento.

```
1 print (mi_tupla[1])      #Devuelve 15
2 print (mi_tupla)        # Devuelve ('Cadena de texto', 15, 2.8, 'otro dato', 25)
```

Conjuntos

Podemos también crear un conjunto

```
1 >>> s={1, 2, 3}
2 >>> print(s)
3 {1, 2, 3}
4 >>> s.add(4)
5 >>> s
6 {1, 2, 3, 4}
7 >>> s.add(2)
8 >>> s
9 {1, 2, 3, 4}
10 >>> s.discard(2)
11 >>> s
12 {1, 3, 4}
13 >>> s.remove(4)
14 >>> s
15 {1, 3}
16 >>> s.discard(4)
17 >>> s.remove(4)
18 Traceback (most recent call last):
19   File "<pyshell#16>", line 1, in <module>
20     s.remove(4)
21   KeyError: 4
```

Diccionarios

A diferencia de las listas y las tuplas, los diccionarios permiten utilizar una clave para declarar y acceder a un valor.

```
1 mi_diccionario = {'clave_1': 12, 'clave_2': 10}
2 print (mi_diccionario['clave_1'])           # Devuelve 12
```

Un diccionario permite eliminar cualquier elemento,

```
1 del (mi_diccionario['clave_2'])
2 print (mi_diccionario)                     # Devuelve {'clave_1': 12}
```

Al igual que las listas permite modificar los elementos,

```
1 mi_diccionario['clave_1'] = 24
2 print (mi_diccionario)                     #Devuelve {'clave_1': 24}
```

5.3.6 Estructuras de control

Una estructura de control es un bloque de código que permite agrupar instrucciones de manera controlada. Las principales estructuras de control son de dos tipos:

- Estructuras de control condicionales.
- Estructuras de control iterativas.

Estructuras de control condicionales

Para evaluar más de una condición simultáneamente se utilizan los operadores lógicos presentados anteriormente.

Las estructuras de control condicionales, se definen mediante el uso de tres palabras reservadas *if* , *elif* y *else* .

Ejemplos:

```
1
2     >>> def cruzar(semaforo):
3         ...             if semaforo == "verde":
4         ...                 print ("Cruzar la calle")
```

```
5         ...             else:
6         ...                 print ("No cruzar la calle")
7
8     >>>def decidir_pago (compra)
9     ...     if compra <= 100:
10    ...         print ("Pago en efectivo")
11    ...         elif compra > 100 and compra < 300:
12    ...             print ("Pago con tarjeta de débito")
13    ...         else:
14    ...             print ("Pago con tarjeta de crédito")
15
```

Estructura múltiple condicional match case

Hasta la versión 3.10, Python no tenía una característica que implementara lo que hace la instrucción *switch* en otros lenguajes de programación. En su lugar, tendrías que usar la palabra clave *elif* para ejecutar múltiples declaraciones condicionales. A partir de la versión 3.10, Python ha implementado una característica de *switch case* llamada *emparejamiento de patrones estructurales*. Se puede implementar esta característica con las palabras clave *match* y *case*. Aquí se muestra un ejemplo de cómo se ve en Python 3.10:

Ejemplo:

```
1
2 lang = input("¿Qué lenguaje de programación quieres aprender? ")
3 match lang:
4     case "JavaScript":
5         print("Puedes convertirte en un desarrollador web.")
6     case "Python":
7         print("Puedes convertirte en un científico de datos")
8     case "PHP":
9         print("Puedes convertirte en un desarrollador backend")
10    case "Solidity":
11        print("Puedes convertirte en un desarrollador de Blockchain")
12    case "Java":
13        print("Puedes convertirte en un desarrollador de aplicaciones móviles")
14    case _:
15        print("El lenguaje no importa, lo que importa es resolver problemas.")
16
```

Listing 45. Ejemplo de match case en Python.

En este código, *match* es la palabra clave que inicia la declaración del *switch*, y *case* se usa para definir cada caso posible. Si ninguno de los casos coincide con el

valor de *lang*, se ejecutará el bloque de código después de case *_*;, que es el caso predeterminado.

Estructuras de control iterativas

En Python se dispone de dos estructuras cíclicas:

while Ejecuta una misma acción mientras una determinada condición se cumpla. Ejemplo:

```
1 >>> def imp_anios(anio):
2     ...     while (anio <= 2012):
3     ...         print "Informes del año", str (anio)
4     ...         anio += 1
5     ...
```

Al probar este programa teniendo como dato de entrada *anio* = 2009, se obtiene:

```
1 >>> imp_anios(2009)
2 Informes del año 2009
3 Informes del año 2010
4 Informes del año 2011
5 Informes del año 2012
```

Con la última línea del programa *anio* + = 1, estamos incrementando en uno la variable *anio*. Esto hace que el ciclo en algún momento termine. Si ocurriera que el valor que se evalúa para el ciclo no es un valor numérico que no puede incrementarse; en ese caso, podremos utilizar una estructura de control condicional, anidada dentro del ciclo, y frenar la ejecución cuando el condicional deje de cumplirse, con la palabra clave reservada *break*:

Ejemplo:

```
1 >>> def edad():
2     ...     while True:
3     ...         edad = input("Edad: ")
4     ...         if int(edad)>100:
5     ...             break
6
7 >>> edad()
8 Edad: 4
```

```
9 Edad: 5
10 Edad: 101
11 >>>
```

Este programa continuará hasta que el usuario introduzca su nombre.

for El ciclo *for*, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla.

Ejemplo:

```
1 >>> mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
2         >>> for nombre in mi_lista:
3             ...     print nombre
4             ...
5
```

Este programa devuelve como resultado:

```
1     Juan
2         Antonio
3         Pedro
4         Herminio
```

Otra forma de iterar con el *for* es la siguiente:

```
1 >>> for anio in range(2001, 2009):
2     ...     print "Informes del Año", str(anio)
3     ...
```

Instrucciones de control de ciclos Como en otros lenguajes, se tienen instrucciones para modificar el comportamiento del flujo de ejecución, ya sea para saltar o detener la ejecución de un ciclo:

- *break*. Dada una condición, la instrucción *break* detiene la ejecución y salta el flujo fuera del ciclo.
- *continue*. Dada una condición, el flujo salta al inicio del ciclo y permite la siguiente iteración, si la condición del ciclo sigue siendo verdadera.

5.3.7 Entrada y Salida básica en Python

Las funciones principales de entrada y salida de datos son:

- La función `print()` es interna del lenguaje Python y se utiliza para imprimir en la pantalla. Para concatenar varios datos a imprimir en pantalla se utilizan comas, e.g. `print("Hola", alguien, "!")`.
- La función `a= input ("Introduzca un valor")`³ despliega un mensaje en pantalla para solicitar un dato que será almacenado en la variable `a` como texto. La función `input`, devuelve el valor ingresado por teclado tal como se escribe.

```
1 >>> variable=input("Edad:")
2 Edad:45
3 >>> variable
4 '45'
5 >>> type(variable)
6 <class 'str'>
7 >>> int(variable)
8 45
9
10 >>> nombre=input("Nombre:")
11 Nombre:carlos a
12 >>> nombre
13 'carlos a'
```

Ejemplo:

```
1 s=" ;Hola, Mundo! "
2 print(s)
3 print(s[1])
4 print(s[7:12])
5
6 # elimina caracteres en blanco del lado izquierdo y derecho de la cadena
7 print(s.strip())
8 print(len(s))
9 print(s.lower())
10 print(s.upper())
11
12 #sustituye "l" por "j"
```

³Antes de Python 3 era llamada `raw_input`

```
13 print(s.replace("l", "j"))
14
15 #separa una cadena y regresa una lista de cadenas
16 str = "Un ejemplo de string....!"
17 print (str.split( ))
18 print (str.split('e',1))
19 print (str.split('e'))
20
21 print("Nombre:")
22 n=input()
23 print("Hola, "+n)
24
25
```

Listing 46. Ejemplo de entrada en Python.

5.3.8 Funciones

Aunque ya hemos usado funciones en Python no se han explicado formalmente.

Sintaxis

```
def <nombre_función> ( [<parametros>] ) :
    <cuerpo de la función>
    [return <expresión>]
```

Como puede verse la instrucción de retorno es opcional. En caso de no regresarse nada explícitamente, se regresa *None*.

Valores por omisión en parámetros

Un argumento por omisión es un parámetro que asume un valor si el valor no es proporcionado en la el argumento de la llamada de la función.

```
1 # Ejemplo de valores por omisión
2
3 def fun(x, y=50):
```



```
4     print("x: ", x)
5     print("y: ", y)
6
7 fun(10)
```

Listing 47. Ejemplo valores por omisión en Python.

Al igual que en C++, los valores por omisión deben estar a la extrema derecha en la lista de argumentos.

Importante

Hay que tener en cuenta que el ligado a los valores por omisión ocurre en la definición de la función.

Usualmente este comportamiento no es el deseado:

```
1
2 def f(x = None):
3     if x is None:
4         x = []
5     x.append(1)
6     return x
7
8 print(f())
9 print(f())
10 print(f())
11 print(f(x = [9, 9, 9]))
12 print(f())
13 print(f())
14
```

Listing 48. Ejemplo en Python.

Palabras clave de argumentos intercambiables

Este concepto implica poner el nombre del argumento en la llamada con sus valores, de manera que no se tenga que recordar el orden de los parámetros.

```
1 def estudiante(nombre, apellido):
2     print(nombre, apellido)
3
4     # Argumentos clave
5     estudiante(nombre='Ada', apellido='Lovelace')
6     estudiante(apellido='Turing', nombre='Alan')
7
```

Listing 49. Ejemplo palabras clave de argumentos en Python.

Argumentos de cantidad variable

En Python, podemos tener argumentos de cantidad variable usando * o **. En el primer caso para argumentos tradicionales sin palabra clave y el segundo caso para argumentos con palabras clave (ver ejemplo anterior).

Argumentos de longitud variable sin palabras clave

```
1
2 # *argvar como variable para los argumentos
3
4 def fun(*argvar):
5     for arg in argvar:
6         print(arg)
7     print(type(argvar)) # argvar es una tupla
8
9 fun('Hola', 'Bienvenido', 'a', 'POO')
10
```

Listing 50. Ejemplo Argumentos de longitud variable sin palabras clave en Python.

```
1
2 # **kwargs como variable para los argumentos con palabras clave
3
4 def fun(**kwargs):
5     for clave, val in kwargs.items():
6         print(f"{clave} == {val}")
7     print(type(kwargs)) #kwargs es un diccionario
8
9 fun(uno='Prueba', dos='de', tres='argumentos')
10
```

Listing 51. Ejemplo Argumentos de longitud variable con palabras clave en Python.

Funciones de primera clase in Python

Los **objetos de primera clase** en un lenguaje son manejados uniformemente. Pueden ser usados dentro de una estructura de datos, pasados como argumentos, o usados en estructuras de control. Un lenguaje como Python que soporta **funciones de primera clase** es aquel que trata a sus funciones como objetos de primera clase. Las funciones de primera clase en Python:

- Una función es una instancia de la clase *function* y *object*.
- Una función puede asignarse a una variable.
- Una función puede pasarse como parámetro a otra función.
- Se puede regresar una función como valor de retorno de una función.
- Se puede almacenar una función en estructuras de datos, por ejemplo en una lista.

Funciones como objetos

```
1
2 #Funciones como objetos
3
4 def saludar(texto):
5     return texto.upper()
6
7 print (saludar('Hola'))
8
9 gritar = saludar
10
11 print (gritar(';Hola!'))
12
```

Listing 52. Ejemplo funciones como objetos en Python.

Funciones como argumentos de otras funciones También conocidas como funciones de orden superior.

```
1
2 def gritar(text):
```

```
3         return text.upper()
4
5     def susurrar(text):
6         return text.lower()
7
8     def saludar(func):
9         # almacenando la función en una variable
10        saludar = func("""Hola, hemos creado una función
11        pasada como un argumento.""")
12        print (saludar)
13
14    saludar(gritar)
15    saludar(susurrar)
16
```

Listing 53. Ejemplo de función como argumento de otra función en Python.
Función regresando a otra función

```
1
2     def crea_sumar(x):
3         def sumar(y):
4             return x+y
5
6         return sumar
7
8    suma_15 = crea_sumar(15)
9
10   print (suma_15(10))
11
12   print (suma_15(15))
13
```

Listing 54. Ejemplo de función regresando a otra función en Python.

5.3.9 Módulos en Python

En Python cada uno de los archivos *.py* se denominan módulos.

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario importar los módulos que se quieran utilizar. Para importar un módulo, se utiliza la instrucción *import*, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el *.py*) que se desee importar.

Importación de un módulo completo:

Se puede importar un módulo completo utilizando la palabra clave *import* seguida del nombre del módulo. Por ejemplo:

```
import modulo
```

Esto carga todo el contenido del archivo "modulo.p" en el programa actual. Luego, se puede acceder a las funciones, variables y clases definidas en ese módulo utilizando la notación de punto, como modulo.funcion(), modulo.variable, o modulo.Clase.

Importación de módulos con alias:

Se puede asignar un alias al módulo importado para hacer que el código sea más legible o evitar conflictos de nombres. Para hacerlo, se utiliza la palabra clave *as*. Por ejemplo:

```
import modulo as alias
```

Esto permite utilizar alias en lugar de modulo para acceder a sus elementos.

Importación selectiva:

Se puede importar solo partes específicas de un módulo utilizando la declaración *from*. Por ejemplo:

```
from modulo import funcion, variable
```

Esto importará solo la función "funcion" y la variable "variable" del módulo "modulo.py". Luego, se pueden utilizar directamente sin el prefijo del módulo.

Importación de todos los elementos:

También se pueden importar todos los elementos de un módulo utilizando el asterisco *, pero esto no se recomienda en general debido a que puede hacer que el código sea menos legible y propenso a conflictos de nombres. Por ejemplo:

```
from modulo import *
```

Esto importará todas las funciones, variables y clases definidas en *modulo.py*.

Es importante recordar que Python buscará los módulos en los directorios especificados en la variable de entorno *sys.path*. Se debe asegurar que el módulo que se intenta importar se encuentre en alguno de los directorios de esta lista o en el directorio actual del script que se está ejecutando.

Ejemplo:

```
1 import modulo    # importar un módulo que no pertenece a un paquete
2 import paquete.modulo1  # importar un módulo que está dentro de un paquete
3 import paquete.subpaquete.modulo1
4
5 from math import sqrt, sin, cos
```

Además, Python tiene sus propios módulos, los cuales forman parte de su biblioteca de módulos estándar, que también pueden ser importados.

A su vez, los módulos pueden agruparse formando paquetes.

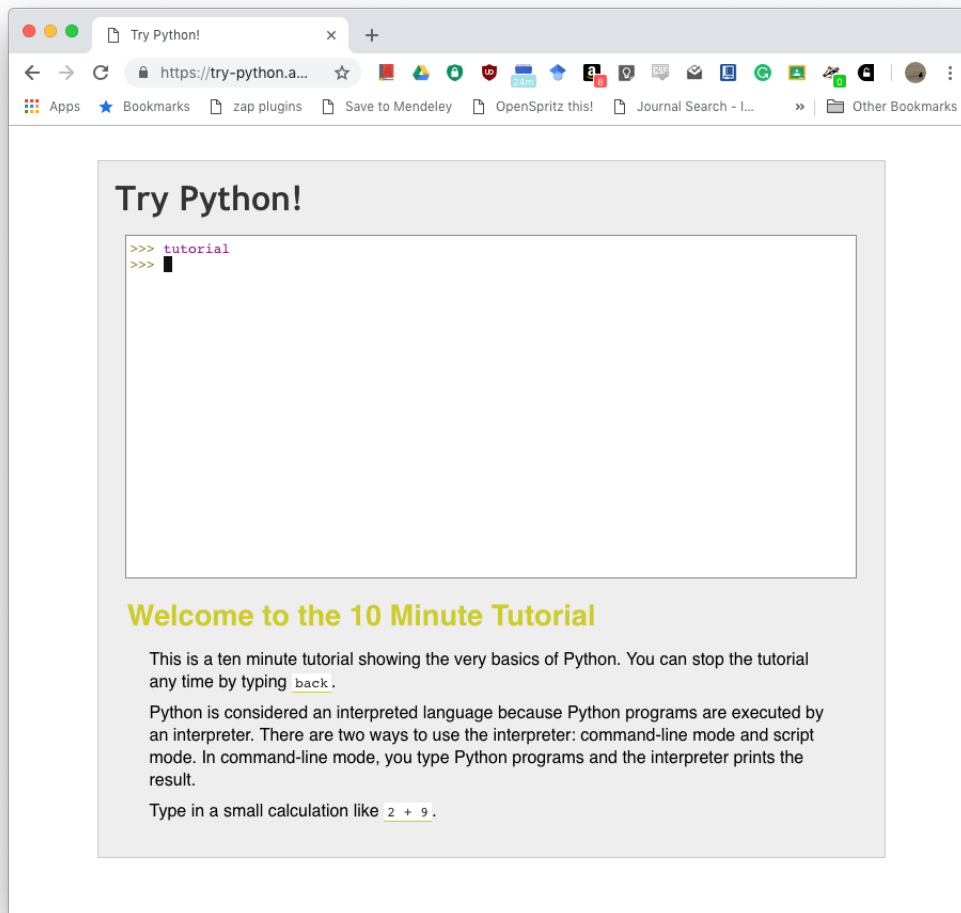
5.3.10 Paquetes en python

Un paquete es una carpeta que contiene archivos *.py*. Pero para que una carpeta pueda ser considerada un paquete, esta debe contener un archivo de inicio *__init__.py*. Este archivo no necesita contener ninguna instrucción, de hecho puede estar completamente vacío. Sin embargo, considerar que este archivo es invocado cuando el paquete es importado por lo que puede contener código necesario para la inicialización del paquete.

Además dentro de los paquetes pueden estar contenidos otros subpaquetes y los módulos no necesariamente pueden estar en un paquete.

5.3.11 Probando Python

¿Por qué no empezar a probar python siguiendo el tutorial en línea? Éste se encuentra disponible en : <https://try-python.appspot.com/>



5.3.12 Python estáticamente tipado

A partir de Python 3.5, se introdujo la funcionalidad de tipado estático en Python a través del uso de anotaciones de tipo. Esto permite al desarrollador especificar el tipo de una variable o parámetro en su declaración, lo que ayuda a prevenir errores de tipo en tiempo de ejecución.

Las anotaciones de tipo se escriben como una expresión después de una variable o parámetro, precedida por un dos puntos (:), por ejemplo:

```
1 def saludar(nombre: str) -> str:
2     return "Hola, " + nombre
```

En este caso, se está declarando que la variable 'nombre' es de tipo *string*, y la función 'saludar' retorna un *string*.

Otro ejemplo:

```
1
2 def f(n: int = 10) -> int:
3     return n*n
4 print("Buen uso", f(8))
5 print("Mal uso", f(.9))
6 print(f())
```

El código anterior corre sin problema a pesar de pasar un valor flotante donde se espera un entero.

Es importante mencionar que las anotaciones de tipo son solo informativas y no se realiza ninguna comprobación en tiempo de ejecución, para comprobar el tipo de una variable o parámetro se requiere de una biblioteca externa llamada 'mypy'.

Capítulo 6

Introducción a C#

6.1 Introducción

C# (C *Sharp*) fue presentado en el verano del 2000 en su versión 1.0, junto con el framework .NET por Microsoft. Aunque la versión final del compilador no fue liberada hasta enero del 2002. La idea de .NET es la de proporcionar una nueva plataforma para el desarrollo de aplicaciones para Windows, independiente del lenguaje¹. Sin embargo, C# siempre ha sido considerado el lenguaje representativo de .NET.

En la práctica se dice que el lenguaje es similar a C++ y Java. Uno no puede negar, que C# fue lanzado en el mejor momento de Java, cuando este lenguaje iba ganando muchos adeptos en la academia y las empresas.

James Gosling (el desarrollador de Java) ha llamado a C# una imitación de Java con lo que Anders Hejlsberg (el líder del proyecto de C#) no está de acuerdo y dice que es más parecido a C++. C# es un lenguaje de programación multiparadigma que es capaz de trabajar bajo los siguientes paradigmas : estructurado, imperativo, orientado a objetos, funcional, genérico, orientado a componentes, *task-driven*, *event-driven* reflexivo concurrente y declarativo.

El origen de su nombre es un dato curioso pues originalmente se iba a llamar *Cool* (C-like Object Oriented Language) pero por cuestiones de derechos fue llamado C#, el cual proviene de la nota musical “C-Sharp” (C sostenido) que indica que la nota escrita es un semitono más alta (o mejor). En comparación con C++ donde ++ significa que C fue incrementado o mejorado en uno, C# se podría interpretar como C++ ++ y # representa los 4 signos + implicando que se trata de un incremento o mejora de C++.

Aunque tuvo un lento arranque, no se puede negar que C# ha ido adquirien-

¹Programa en cualquier lenguaje y corre en Windows es la idea. Contrario a la propuesta de Java, de compila una vez, corre donde sea. Sin embargo, ambos conceptos están convergiendo. JVM ahora ejecuta otros lenguajes como Scala y .NET tiene una versión libre (Mono).

do fuerza con el tiempo y se ha constituido en una opción real de desarrollo de software desde hace un buen tiempo.

6.2 Herramientas

Es posible usar una versión libre de C#, el proyecto Mono, que ofrece un framework multiplataforma de .NET. Existen versiones de este framework para Linux, Mac OS X y Windows entre otros.

Puede instalarse únicamente las herramientas para la terminal e integrarlas al IDE de su preferencia (o usar la consola). También se tiene un IDE llamado *Mono-Develop*. Para más información ver:

<http://mono-project.com/>

Una vez instalado Mono, debe poder ejecutarse el compilador:

```
$mcs < programa.cs >
```

producirá código .NET 1.1

```
$gmcs < programa.cs >
```

generará código .NET 2.0

El compilador generará un ejecutable .exe. Ejecutar una aplicación desde mono implicaría usar la máquina de mono:

```
$mono < programa.exe >
```

En Windows también podría ejecutarse directamente, como una aplicación ejecutable, aunque dependiente del *runtime* de .NET

6.3 Fundamentos de C#

6.3.1 Literales

C# tiene caracteres denominados secuencias de escape para facilitar la escritura con el teclado de símbolos que carecen de representación visual.

Enteros: decimal, hexadecimal, entero largo, entero largo sin signo Coma flotante: float, double, decimal. Caracteres: char String: string

6.3.2 Variables

Las variables son identificadores asociados a valores. Se declaran indicando el tipo de dato que almacenará y su identificador. Un identificador puede:

- Empezar por " ".
- Contener caracteres *Unicode* en mayúsculas y minúsculas.

Un identificador no puede:

- Empezar por un número.
- Empezar por un símbolo, ni aunque sea una palabra clave.
- Contener más de 511 caracteres.

También se puede declarar una variable sin especificar el tipo de dato, mediante la palabra *var*.

6.3.3 Constantes

Las constantes son valores inmutables, y por tanto no se pueden cambiar. Estas se declaran con la palabra clave *const*.

Otra tipo es *readonly*, no requiere de asignación al mismo tiempo que se declara.

6.3.4 Operadores

Los operadores aritméticos funcionan igual que en C y C++. El resultado de los operadores relacionales y lógicos es un valor booleano. Los operadores a nivel de bits no se pueden aplicar a tipos *bool*, *float*, *double* o decimal.

- Aritméticos
- Lógicos
- A nivel de bits
- Concatenación
- Incremento, decremento
- Desplazamiento
- Relacional
- Asignación
- Acceso a miembro
- Indexación
- Conversión
- Condicional
- Creación de objeto
- Información de tipo

6.3.5 Estructuras de control

Las estructuras de control *if*, *switch*, *while*, *do-while* y *for* son básicamente las mismas de lenguajes estilo C/C++ y Java.

El *for* permite la declaración de variables en su inicialización, a diferencia del estándar actual de C++.

Agrega un *foreach* que funciona de manera similar a la variación de *for* en Java pero cambiando el nombre de la instrucción al ya mencionado:

```
foreach (int x in intList) { ... }
```

6.3.6 Entrada y Salida básica

Salida básica a consola puede hacerse con los métodos de consola *WriteLine()* y *Write()*.

```
1 using System;
2 namespace Ejemplo
3 {
4     class Test {
5         public static void Main(string[] args) {
6             Console.WriteLine("C# es cool");
7         }
8     }
9 }
```

Listing 55. Ejemplo.

La entrada simple por consola puede hacerse con los métodos *ReadLine()*, *Read()* y *ReadKey()*. *Readline()* recibe la siguiente línea en un flujo de entrada estándar y regresa dicha cadena. *Read()* recibe el siguiente carácter del flujo de entrada estándar y regresa el carácter ASCII. Finalmente, *ReadKey()* obtiene la siguiente tecla presionada por el usuario.

```
1 using System;
2 namespace Ejemplo
3 {
4     class Entrada {
5         public static void Main(string[] args) {
6             int entradaUsuario;
7
8             Console.WriteLine("Presiona una tecla para continuar...");
9             Console.ReadKey();
10        }
```

```
10         Console.WriteLine();  
11         Console.Write("Leyendo entrada con Read(): ");  
12         entradaUsuario = Console.Read();  
13         Console.WriteLine("Valor ASCII = {0}", entradaUsuario);  
14     }  
15 }  
16 }
```

Listing 56. Ejemplo.

Capítulo 7

Introducción a Scala

7.1 Introducción

Scala es un lenguaje con características funcionales y orientadas a objetos que se ejecuta sobre la máquina virtual de Java y tiene gran interacción con éste lenguaje.

¿Es Scala orientado a objetos o funcional? En realidad se trata de una nueva generación de lenguajes multiparadigma. Puede considerarte 100 % orientado a objetos pero incluye también un buen número de características del paradigma funcional. **Scala es multiparadigma y multiplataforma.**

Fue creado por Martin Odersky, quien inició el proyecto en 2001 liberando la primera versión en 2003. Odersky trabajó desde 1995 con la máquina virtual de java y lideró el desarrollo de *javac* de la versión 1.1 a la 1.4 [?].

Odersky y Phillip Wader también trabajaron en un lenguaje llamado Pizza que trabajaba con la JVM y contenía clases genéricas entre otras características. Pizza evolucionó para proporcionar clases genéricas en Java.

Scala esta disponible en www.scala-lang.org . Iniciado como proyecto académico ha encontrado acogida en la industria. Por ejemplo, Twitter fue originalmente desarrollado en Ruby, pero después movió partes de su código a Scala.

Actividad: lectura de artículo

https://www.computerworld.com.au/article/315254/-z_programming_languages_scala

7.2 Herramientas

Para ejecutar Scala debe tenerse instalada la JVM. Scala puede ser obtenido del sitio mencionado anteriormente.

Scala es accesible desde la consola de comandos tecleando *scala*. Deberias de ver una introducción como:

```
Welcome to Scala version 2.7.3.final (Java HotSpot(TM) 64-Bit Server VM,
1.6.0_07).
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Scala puede ejecutar código de tres formas:

1. Interactivamente
2. Ejecución interpretada de archivos
3. En modo compilado en archivos de clases como en Java

Scala puede ser iniciado en su modo interactivo simplemente ejecutando *scala* desde una terminal del sistema¹:

```
$scala
Welcome to Scala version 2.7.5.final (Java HotSpot(TM) Client VM, Java 1.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Como cualquier intérprete, es posible empezar ejecutar expresiones directamente:

```
1 scala> 5+10
2 res0: Int = 15
3
4 scala> res0 * 2
5 res1: Int = 30
6
7 scala> val hola= "Hola Mundo!"
8 hola: java.lang.String = Hola Mundo!
9
10 scala> print ("Hola Mundo!")
11 Hola Mundo!
```

Recordar que Scala se basa en Java, por lo que es simple importar las bibliotecas de la API de Java:

¹Asumiendo que se encuentra instalado y accesible desde cualquier directorio.


```
1 scala> val fecha= new Date
2 fecha: java.util.Date = Mon Aug 31 13:43:32 CDT 2009
```

Por otro lado, la ejecución de scripts se hace utilizando el mismo intérprete, pero indicando, en el momento de ejecución, el nombre del programa:

```
>scala programa.scala
```

Finalmente, como se mencionó, es posible compilar los programa en Scala y obtener archivos *.class*, usando el programa *scalac*:

```
>scalac programa.scala
```

Se necesita que los archivos fuente contengan una o más clases definidas.

También es posible compilar usando *fsc*. *fsc* (*fast Scala compiler*) es un compilador que se queda como proceso corriendo, esperando por mas compilaciones, ayudando a realizar múltiples compilaciones en un menor tiempo, pero usando por lo tanto mas recursos al permanecer en ejecución.

7.3 "Hola, Mundo"

Veamos ahora el clásico "Hola, Mundo.^{en} Scala:

```
1 object HolaMundo {
2   def main(args: Array[String]) = {
3     println("Hola, Mundo!")
4   }
5 }
```

Listing 57. Ejemplo "Hola, Mundo.^{en} Scala.

Otra posibilidad es definir un objeto que extienda *App*²:

```
1 object HolaMundo extends App{
2   println("Hola, Mundo!")
3 }
```

Listing 58. Ejemplo 2 de "Hola, Mundo.^{en} Scala.

²Ver: [How to launch a Scala application with an object \(main, app\)](#)

Tipo	Valor mínimo	Valor máximo
Long	-2^{63}	$2^{63} - 1$
Int	-2^{31}	$2^{31} - 1$
Short	-2^{15}	$2^{15} - 1$
Char	0	$2^{16} - 1$
Byte	-2^7	$2^7 - 1$

Cuadro 7.1. Literales en Scala

7.4 Fundamentos de Scala

7.4.1 Convenciones léxicas

Scala utiliza la misma convención de comentarios de Java, C#, C, etc.; es decir utiliza `//comentario` y `/* comentario */`. Scala utiliza la regla de **mayor coincidencia**, que se refiere al uso de paréntesis en los números:

```
(1) . ( ( (2) . * (3) ) . / (x) )
```

ya que 1. es una coincidencia válida y es mayor que 1, haciendo que este sea un *Double* y no un *Int* y al tener `(1) . + (2)` se consideran enteros tanto 1 como 2.

7.4.2 Literales

Existen diferentes valores literales como enteros, punto flotante, booleanos, caracteres, cadenas, símbolos, funciones, tuplas, etc. Las literales enteros pueden ser expresadas en decimal, hexadecimal u octal. Los límites de estas literales se muestran en la siguiente el Cuadro 7.1

Los valores flotantes pueden llevar la letra *f* o *F* al final del valor para indicar que son de este tipo, también pueden tener la letra *d* o *D* para indicar que son *Double*, y pueden utilizar la letra *e* o *E* de exponencial para escribir un número en notación científica, por ejemplo: `.14`, `3.14f`, `3.14F`, `3.14d`, `3.14D`, `3.14e+5`, `3.14e-5`, `3.14e+5f`, `3.14e-5F`, etc. Los valores tipo flotantes consisten de todos los IEEE 754 de 32-bits mientras que los dobles son de IEEE 754 de 64-bits.

Los literales tipo booleanos constan de *true* y *false*, por ejemplo:

```
1 scala> val b1 = true
2 b1: Boolean = true
3 scala> val b1 = false
4 b1: Boolean = false
```

Secuencia	Significado	Secuencia	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador horizontal	\"	Comillas dobles
\n	Salto de línea	\'	Comillas simples
\f	Salto de página	\\	Barra invertida

Cuadro 7.2. Secuencias de escape en Scala

Un carácter es o bien un carácter *Unicode* o una secuencia escrito entre ' '. Un carácter Unicode entre 0 y 255 también puede estar representado por un octal, es decir, por una barra invertida (\) seguida de una secuencia de hasta tres caracteres octales, por ejemplo: 'A' y '\u0041' ('A' en *Unicode*), o '\n' y '\012' (en octal). En el siguiente cuadro (??) se muestran los caracteres que reconoce Scala.

El *string* es una secuencia de caracteres entre doble comillas o triple. Scala permite utilizar cadenas multi-lineas sin utilizar espacios en blanco extras en la cadena de salida:

```

1 def hello(name: String) = s "Bienvenido!"
2   Hola, $nombre!
3   |Nos alegra verte.
4   | Deja espacio en blanco extra.
5 hello("programa en Scala")

```

Las literales funciones se pueden escribir de dos formas:

```

1 val f1: (Int,String) => = (i,s) => s+i
2 val f2: Function2[Int,String,String] = (i,s) => s+i

```

son funciones que reciben un *Int* y un *String* y devuelven un *String*. También se pueden regresar dos o más valores en un método usando la biblioteca *Tuple* para agrupar 2 elementos, la sintaxis es escribir los elementos dentro de paréntesis separados por comas, por ejemplo:

```

1 val t1: (Int,String) = (1, "Dos")
2 val t2: Tuple2[Int,String] = (1, "Dos")

```

Se puede utilizar la sintaxis literal para construir la variable *t* y poder tener acceso a cada uno de los elementos de la siguiente forma:

```

1 val t = ("Hola",1,3.4)
2 println("Tupla completa: " + t)
3 println("Primer elemento de la tupla: " + t._1)
4 println("Segundo elemento de la tupla: " + t._2)
5 println("tercer elemento de la tupla: " + t.3)

```

Declarar tres valores *t1*, *t2*, *t3* y asignarlos a una tupla:

```

1 val (t1,t2,t3) = ("Hola","!",0x22)
2 println("Tupla: " + t1 + ", " + "t2" + ", " + t3)

```

O se puede construir una tupla a partir de otra:

```

1 val (t4,t5,t6) = ("Hola",1,0x22)
2 println("Tupla: " + t4 + ", " + "t5" + ", " + t6)

```

Otra forma de definir una tupla es con `→`, por ejemplo `(1,Uno)` es lo mismo que `1 → "Uno"`.

7.4.3 Variables

Scala utiliza dos tipos de variables, inmutables o mutables. Las variables inmutables solo pueden ser de lectura mientras que las variables mutables son de lectura y escritura. Las variables inmutables son declaradas con la palabra clave *val*:

```

1 scala> val array: Array[String] = new Array(5)
2   array: Array[String] = Array(null,null,null,null,null)

```

Scala es como Java, ya que la mayoría de las variables son referenciados a objetos *head-allocated*, por lo que la referencia del arreglo no se puede cambiar para apuntar a un arreglo diferente, sin embargo, los elementos del arreglo si son mutables. Si se trata de modificar el tamaño del arreglo después de ser inicializada entonces el programa marcará un error:

```
array = new Array(2)
```

Los elementos de una variable inmutable debe ser inicializada en el momento que se declara. Pero se pueden modificar sus valores después de ser inicializada:

Operador	Operación
&&	and
	or
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que
==	igualdad
!=	diferente

```
scala>array(0)=3.23
```

Las variables mutables se declaran con la palabra clave *var* y debe ser inicializada inmediatamente después de ser declarada:

```
1 scala> var variable: Double = 100.0
2 variable: Double = 100.0
```

Puede darse la excepción de que las variables mutables o inmutables no sean declaradas en el momento, pero es cuando se utiliza como constructor de una clase.

7.4.4 Operadores

Scala utiliza los operadores condicionales de Java y se pueden observar en la siguiente tabla:

La mayoría de los operadores se comportan como lo hacen en Java y otros lenguajes. Sin embargo los operadores `==` y `!=` en java comparan sólo referencias a objetos y realiza una comprobación de igualdad lógica, pero Scala aparte de realizar una comparación lógica también llama al método `igual`.

7.4.5 Estructuras de control

Condicional *if*

La condicional es superficialmente muy similar al de Java, hace la comparación lógica y realiza el bloque correspondiente. Sin embargo, casi todas las declaraciones en Scala son expresiones que devuelven valores, por lo que se puede asignar el resultado a una expresión:

```
1 val configFile = new java.io.File("somefile.txt")
2 val configFilePath = if(configFile.exists()){ configFile.getAbsolutePath() }
```

```
3 }else{
4     configFile.createNewFile()
5     configFile.getAbsolutePath()
6 }
```

Condicional *match*

Pattern match o coincidencia de patrones es una estructura similar al switch de Java para ser usado en lugar de una serie de estructuras condicionales if.

```
1
2 import scala.util.Random
3
4 val x: Int = Random.nextInt(10)
5
6 x match {
7     case 0 => "cero"
8     case 1 => "uno"
9     case 2 => "dos"
10    case _ => "otro"
11 }
```

Recordemos que en Scala todo puede ser considerado una expresión. Si por un case se requiere más de una línea, todas las líneas se consideran un bloque sin necesidad de usar llaves para indicar el bloque.

Ciclo *for*

Se puede recorrer un arreglo de tal manera:

```
1 val dogBreeds = List ("Doberman", "Dachshund", "Great Dane", "Scottish Terrier")
2 for (breed <- dogBreeds)
3     println(breed)
```

Es decir, para cada elemento de la lista *dogBreeds* se crea una variable temporal llamada *breed* con el valor del elemento y lo muestra en pantalla. Al operador `<-` se le conoce como expresión generadora. También se puede utilizar un rango de la forma tradicional:

```
for(i<-1 to 10)println(i)
```

Ciclo *while* y *do-while*

El ciclo *while* y *do-while* son parecidos al de Java, realiza cierta tarea siempre que la condición sea verdadera, por ejemplo:

```
1 import java.util.Calendar
2 def isFridayThirteen(cal: Calendar): Boolean = {
3     val dayOfWeek = cal.get(Calendar.DAY_OF_WEEK)
4     val dayOfMonth = cal.get(Calendar.DAY_OF_MONTH)
5     (dayOfWeek == Calendar.FRIDAY) && (dayOfMonth == 13)
6 }
7 while(!isFridayThirteen(Calendar.getInstance())) {
8     println("Today isn't Friday the 13th. Lame.")
9     Thread.sleep(86400000)
10 }
```

7.4.6 Entrada y Salida básica

Módulos ¿?

Bloques de código

Como en Scala todo puede ser considerado una expresión. Inclusive un bloque de código regresa un resultado, dicho bloque de código entonces puede ser usado por un método o una variable

Capítulo 8

Introducción a D

8.1 Introducción

Es un nuevo lenguaje de propósito general altamente influenciado por C++. Sin embargo, D es de más alto nivel pero se dice que conserva un rendimiento similar a C++ y ofrece la productividad de lenguajes como Ruby y Python. Creado por **Walter Bright** para la compañía **Digital Mars**.

A diferencia de la nueva ola de lenguajes, D no es un lenguaje tipo *script* ni incluye una máquina virtual para su ejecución. Sigue permitiendo acceso directo a las APIs del sistema operativo y al hardware.

D se puede considerar una reingeniería de C++ e influenciado por lenguajes como Java, Eiffel. D es un lenguaje de programación multiparadigma. Incluye paradigmas: Imperativo, Orientado a objetos, Funcional y Concurrente.

Actividad: lectura de artículo científico

http://www.computerworld.com.au/article/253741/-z_programming_languages_d

D es uno de los lenguajes usados por Facebook a partir de 2013, en sustitución de C++¹.

D, a diferencia de C++, utiliza una sintaxis y unas construcciones mucho más sencillas y lógicas. El rendimiento de C++, uno de sus puntos fuertes, también se ve reflejado en D y en algunas ocasiones incluso lo supera.

Características de D:

Gestión automática de memoria(recolección de basura): esto quiere decir que el programador seguirá creando los nuevos objetos con *new* pero ya no tendrá que preocuparse de borrarlos con *delete* porque existirá un **recolector de basura** que se encargará de eliminar automáticamente los objetos para los que ya no exista ninguna referencia. Sin embargo, si el programador lo desea, la recolección de basura

¹<http://forum.dlang.org/thread/l37h5s2gd81@digitalmars.com>

puede ser controlada: los programadores pueden agregar y excluir rangos de memoria de ser observados por el recolector, pueden pausar y reanudar el recolector y forzar un ciclo generacional o de recolección completa.

- **Gestión de errores mediante manejo de excepciones:** el sistema de manejo de excepciones es superior al de C++ al incorporar algunas características de lenguajes más recientes. Cuando una excepción producida en código D no se captura, se muestra un mensaje de error con información de la excepción.
- **Guardias de ámbito para asegurar la ejecución de código a la salida de un ámbito:** hay recursos como los archivos, los cerrojos y *mutex* que se siguen teniendo que liberar manualmente; pero hay otros. Realmente, usando las guardias de ámbito, la instrucción *finally* nunca es necesaria aunque se sigue manteniendo en el lenguaje. Las guardias de inclusión no sustituyen a la instrucción *except*; para capturar excepciones y realizar operaciones en la captura debemos seguir usándola.
- **Estructuración del código en módulos y paquetes:** la estructuración del código y las bibliotecas se hace usando módulos y paquetes. Un módulo no es más que un archivo fuente de código D (generalmente con extensión .d). Al contrario de lo que sucede en otros lenguajes, cuando importamos los símbolos de un módulo no hace falta anteponer el nombre del módulo con un punto antes de llamar a un símbolo.
- **Compatibilidad de llamada con C:** Debido a la difusión del lenguaje, la mayoría de las APIs de sistemas operativos y bibliotecas de sistemas están escritas en C u ofrecen una interfaz para el mismo. D puede acceder a bibliotecas de C. Como los tipos de datos de D suelen tener una correspondencia muy directa con los de C este proceso suele ser bastante sencillo; sin embargo para conversiones de archivos de cabecera .h más complicados se dispone de la herramienta *htod* que realiza la conversión de tipos y sintáxis de forma automática, tomando como entrada un archivo .h de C y generando un archivo .d que podemos incluir en nuestros proyectos.
- **Delegados, funciones anidadas y funciones literales:** existe un tipo de dato llamado **delegado** que puede usarse para pasar referencias a un método de una clase como parámetros para otras funciones y métodos. En concepto son similares a los punteros a método de C++ pero con una sintáxis tanto de declaración como de creación y uso mucho más sencilla. D permite tener funciones anidadas y funciones anónimas. Las funciones anidadas son las que están definidas dentro de otra función y son muy útiles para estructurar nuestro código de una forma más jerárquica. Las funciones anónimas son funciones sin nombre que suelen utilizarse como argumento para una función que espera recibir una función como argumento.

- **Declaración anticipada de funciones innecesaria:** para que una función pueda llamar a otra no es necesario que esta última haya sido declarada con anterioridad a la primera.

D retiene la habilidad de C++ de hacer código de bajo nivel, permitiendo incluir código en ensamblador.

8.2 Herramientas

Actualmente se encuentra accesible la versión 2.x, la cual fue presentada en junio de 2007. Existen diferentes implementaciones del lenguaje. La principal es desarrollada por la misma compañía (Digital Mars) en versiones Mac OSX Windows, Linux y FreeBSD. Otras distribuciones están disponibles, por ejemplo un compilador D.NET, obviamente para .NET

Si se usa la implementación de Digital Mars, entonces el compilador es *dmd*. La extensión usada para archivos de código fuente es *.d*. Existen otras extensiones relevantes (*.dd* para archivos de *Ddoc* – similar a *javadoc*-, *.di* para archivos de interfaces, y *.def* para archivos de definición de módulos, entre otros).

8.3 Fundamentos de D

8.3.1 Convenciones léxicas

Todos los archivos D tendrán extensión *.d*.

Ejemplo:

```
1 import std.stdio;
2 void main(string[] args) {
3     writeln("Hola, Mundo!");
4 }
```

Listing 59. Ejemplo "Hola, Mundo!".^{en D.}

Se importa la biblioteca *io* que ofrece las operaciones básicas de E/S. *writeln* que se utiliza en el programa anterior para la impresión de una línea de texto. El único módulo que utiliza este programa es *std.stdio*, que maneja la entrada y salida de datos.

Los comentarios en D pueden ser multilínea: el cual inicia con */** y cierra con **/*, y los comentarios de una sola línea con *//*

Los identificadores en D son los nombres que se les asigna para distinguir a las variables, funciones o cualquier otro elemento definido por el usuario. Un identi-

Secuencia de escape	Significado
\\	caracter \
\'	caracter '
\"	caracter "
\b	backspace
\f	Form feed
\n	Nueva línea
\r	Return
\t	Tabulador horizontal
\v	Tabulador vertical

ficador comienza con una letra A a la Z, o de la a a la z o un guión bajo (_)seguido de cero o mas letras, subrayado y los dígitos (0 a 9).

D no permite caracteres de puntuación tales como @, \$ y % dentro de los identificadores. D es un lenguaje sensible a mayúsculas y minúsculas.

8.3.2 Literales

Los tipos de valores constantes que forman parte del código fuente son llamados literales. Las literales pueden ser alguno de los tipos de datos básicos y pueden dividirse en Números Enteros, Números punto Flotante, Caracter, *Strings* y Valores Booleanos.

- Las literales enteros pueden ser: Decimal, Octal, Binario, Hexadecimal. Una literal entera también puede tener un sufijo que es una combinación de U o L, de *unsigned* y *Long*. Para escribir una literal hexadecimal integral se usa el prefijo 0x o 0X seguido por una secuencia de letras 0-9, a-f, A-F, o _.
- Literales de Punto Flotante: pueden ser especificados sistema decimal como 1.568 o en sistema hexadecimal. En el sistema decimal, un exponente puede ser representado agregando el carácter E o e seguido del valor del exponente, por ejemplo 2.3e4, aunque también 2.3e4 y 2.3e + 4 son lo mismo (A+ carácter pueden especificarse antes del valor del exponente). Por default la literal de punto flotante es *double*. F o f significa flotante.
- Literal Booleana: Existen dos valores posibles para las literal booleana (*True* y *False*).
- Literal Carácter: encerradas entre comillas simples(' '). Una literal de este tipo puede ser un carácter o una secuencia de escape('\t').

Ejemplo donde se utilizan las secuencias de escape:

```
1 import std.stdio;
2 int main(string[] args){
3     writeln("Hello\tWorld%c\n", '\x21');
4     writeln("Have a good day%c", '\x21');
5     return 0;
6 }
```

Listing 60. Ejemplo en D donde se utilizan las secuencias de escape.

- Literal *String*: son encerradas en comillas dobles, un *String* puede contener caracteres similares a las literales carácter (secuencias de escape, caracteres universales)
- Literal de Arreglo y asociación de arreglo: *Strings* son un particular tipo de arreglos. Una literal de arreglo es representada como una secuencia de valores separada por una coma encerrada por corchetes cuadrados. El tamaño del arreglo es la longitud de la lista separada por comas. El arreglo es no immutable, significa que puede modificarse después de su inicialización.
- Literal de Funciones: en algunos lenguajes, cada función tiene un nombre que se elige en el momento de la definición, subsecuentemente la función es llamada con ese nombre. Otros lenguajes tiene la posibilidad de definir funciones anónimas (funciones lambda). Esta característica vuelve poderoso al lenguaje al usar funciones de orden superior, estas funciones toman como parámetros y/o retornan otras funciones. D cuenta con literal de Funciones para definir funciones anónimas.
-

8.3.3 Variables

Una variable es sólo un nombre dado a un área de almacenamiento que nuestro programa puede manipular. Cada variable en D tiene un tipo específico, el cual determina el tamaño. Tipos básicos de variables en D: *char*, *int*, *float*, *doble*, *void*.

La siguiente tabla detalla los tamaños de almacenamiento de los tipos enteros (*int*):

La siguiente tabla detalla los tamaños de almacenamiento de los tipos de punto flotante:

La siguiente tabla detalla los tamaños de almacenamiento de los tipos carácter (*char*):

El tipo *void* se utiliza en dos situaciones:

Tipo	Tamaño	Rango de valor
bool	1 byte	false o true
byte	1 byte	-128 a 127
ubyte	1 byte	0 a 255
int	4 bytes	-2,147,483,648 a 2,147,483,647
uint	4 bytes	0 a 4,294,967,295
short	2 bytes	-32,768 to 32,767
ushort	2 bytes	0 a 65,535
long	8 bytes	-9223372036854775808 a 9223372036854775807
ulong	8 bytes	0 a 18446744073709551615

Tipo	Tamaño	Rango de valor
float	4 bytes	1.17549e-38 a 3.40282e+38
double	8 bytes	2.22507e-308 a 1.79769e+308
real	10 bytes	3.3621e-4932 a 1.18973e+4932
ifloat	4 bytes	1.17549e-38i a 3.40282e+38i
idouble	8 bytes	2.22507e-308i a 1.79769e+308i
ireal	10 bytes	3.3621e-4932 a 1.18973e+4932
cfloat	8 bytes	1.17549e-38+1.17549e-38i a 3.40282e+38+3.40282e+38i
cdouble	16 bytes	2.22507e-308+2.22507e-308i a 1.79769e+308+1.79769e+308i
creal	20 bytes	3.3621e-4932+3.3621e-4932i a 1.18973e+4932+1.18973e+4932i

Tipo	Tamaño
char	1 byte
wchar	2 bytes
dchar	4 bytes

1. La función devuelve void: funciones que no devuelven nada o que devuelven nulo. Una función sin valor de retorno tiene el tipo de retorno void.
2. Argumentos de la función vacío: Una función que no acepta parámetros, vacío, con ningún parámetro.

Definición de variables en D:

```
1 int i, j, k;  
2 char c, ch;  
3 float f, salary;  
4 double d;
```

Las variables también pueden ser inicializadas desde su declaración:

```
1 extern int d = 3, f = 5;  
2 int d = 3, f = 5;  
3 byte z = 22;  
4 char x = 'x';
```

En este [ejemplo](#) las variables han sido definidas en el tope del programa pero se redefinen e inicializan nuevamente dentro del main:

```
1 import std.stdio;  
2     int a = 10, b = 10;  
3     int c;  
4     float f;  
5     int main () {  
6         writeln("Value of a is : ", a);  
7         /* variable re definition: */  
8         int a, b;  
9         int c;  
10        float f;  
11        /* Initialization */  
12        a = 30;  
13        b = 40;  
14        writeln("Value of a is : ", a);  
15        c = a + b;  
16        writeln("Value of c is : ", c);  
17        f = 70.0/3.0;
```

Tipo	Operadores
Operadores Aritméticos:	+, -, *, /, %, ++, --
Operadores Relacionales:	==, !=, >, <, >=, <=
Operadores Lógicos:	&&, , !
Operadores Bit a bit:	&, y ^
Operadores de Asignación:	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, =
Operadores Misc:	sizeof, &(dirección de una variable), * (apuntador), ? :

```

18         writeln("Value of f is : ", f);
19         return 0;
20     }

```

Listing 61. Ejemplo de alcance en la definición de variables en D.
Con la siguiente salida:

```

Value of a is : 10
Value of a is : 30
Value of c is : 70
Value of f is : 23.3333

```

8.3.4 Operadores

Un operador es un símbolo que llama al compilador para ejecutar una operación específica, lógica o matemática. D provee los siguientes tipos de operadores:

Precedencia de operadores

Ver Cuadro 9.1

8.3.5 Arreglos

Los arreglos en D pueden ser estáticos o dinámicos.

Un arreglo estático se declara de la siguiente forma:

Sintaxis
<tipo> [<tamaño>] <variable>

De tal forma que un arreglo de 10 enteros quedaría de la siguiente forma:

```
int [10] arr
```


Cuadro 8.1. Precedencia de operadores en D

Categoría	Operador
Postfijo	[] -> . ++ - -
Unario	+ ! ~ ++ __ type* & sizeof
Multiplicativo	* / %
Aditivo	+ -
Shift	<< >>
Relacional	<= >=
Igualdad	== !=
Lógicos	AND &&
Lógicos	OR
Condicional	?:
Asignación	= += -= *= /= %= >>= <<= &= ^= =
Coma	,

Un arreglo dinámico por su parte se declararía de manera similar pero sin indicar el tamaño específico.

Sintaxis
<tipo> [] <variable>

Por lo que la declaración de un arreglo dinámico entero sería de la siguiente forma:

```
1 int [] a;
```

Los arreglos son objetos por lo que podemos obtener el número de elementos de un arreglo del atributo *length*.

Ejemplo:

```
1 int[10] a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
2 int[] b1, b2, b3, b4;
3
4 b1 = a;
5 b2 = a[];
6 b3 = a[0 .. a.length];
7 b4 = a[0 .. $];
8 writeln(b1);
9 writeln(b2);
```

```
10 writeln(b3);  
11 writeln(b4);  
12
```

La longitud de un arreglo dinámico se puede hacer asignando un nuevo valor al atributo *length*:

```
1 int [] a;  
2  
3 a.length=10;  
4
```

8.3.6 Estructuras de control

if: una sentencia consiste de una expresión booleana seguida de una o mas declaraciones.

Sintaxis

```
if (<expresión_booleana>){  
    /* se ejecuta si la expresión booleana es verdadera */  
}
```

Ejemplo:

```
1 import std.stdio;  
2 int main (){  
3     /* Definición de variable local */  
4     int a = 10;  
5  
6     if( a < 20 ){  
7         writefln("a es menor que 20" );  
8     }  
9     writefln("valor de a es : %d", a);  
10    return 0;  
11 }
```

Listing 62. Ejemplo de estructura de control *if* en D.

if...else: una sentencia *if* puede estar seguida de una sentencia *else* opcional, que se ejecuta cuando la expresión booleana es falsa.

Sintaxis

```
if (<expresión_booleana>){  
  
} else {  
  
}
```

Ejemplo:

```
1      import std.stdio;  
2      int main () {  
3          int a = 100;  
4  
5          if( a < 20 ) {  
6              writeln("a es menor que 20" );  
7          }  
8          else {  
9              writeln("a no es menor que 20" );  
10         }  
11         writeln("el valor de a es : %d", a);  
12         return 0;  
13     }
```

Listing 63. Ejemplo de estructura *if.. else* en D.

Switch: Permite a una variable ser evaluada con una lista de valores.

Sintaxis

```
switch(<expresión>){  
case <expresión-constante> :  
    <instrucciones>;  
    break; /* opcional */  
case <expresión-constante> :  
    <instrucciones>;  
    break; /* opcional */  
  
    default : /* Opcional */  
        <instrucciones>;  
}
```

```
1 import std.stdio;  
2  
3 int main (){  
4     char grade = 'B';  
5  
6     switch(grade){  
7         case 'A' :  
8             writeln("Excelente!" );  
9             break;  
10        case 'B' :  
11        case 'C' :  
12            writeln("Bien hecho" );  
13            break;  
14        case 'D' :  
15            writeln("Pasaste" );  
16            break;  
17        case 'F' :  
18            writeln("Trata de nuevo" );  
19            break;  
20        default :  
21            writeln("Reprobado" );  
22        }  
23        writeln("Calificación %c", grade );  
24    }
```

```
25     return 0;  
26 }
```

Listing 64. Ejemplo de estructura *switch* en D.

while: repite una(s) instruccin(es) mientras la condición es verdadera. Comprueba la condición antes de el cuerpo del ciclo.

Sintaxis

```
while(<condición>) {  
    <instrucciones>;  
}
```

Ejemplo:

```
1  import std.stdio;  
2  int main ()      {  
3      int a = 10;  
4  
5      while( a < 20 ){  
6          writeln("Valor de a: %d", a);  
7          a++;  
8      }  
9      return 0;  
10 }  
11
```

Listing 65. Ejemplo de estructura *while* en D.

for: ejecuta una secuencia de instrucciones múltiples veces.

Sintaxis

```
for ( <ini>; <condición>; <incremento> ){  
    <instrucciones>;  
}
```

1. `ini` se ejecuta primero y solo una vez.
2. La condición es evaluada. Si es verdadera, el cuerpo del ciclo se ejecuta. Si es falsa el cuerpo del ciclo no se ejecuta y cede el control a la siguiente instrucción después del ciclo.
3. Después de ejecutar el ciclo, se ejecuta la instrucción incremento.
4. La condición es evaluada nuevamente.

Ejemplo:

```
1  import std.stdio;
2  int main () {
3      for( int a = 10; a < 20; a = a + 1 ) {
4          writeln("Valor de a: %d", a);
5      }
6      return 0;
7  }
```

Listing 66. Ejemplo de estructura *for* en D.

do while: similar a *while* con la diferencia que la condición se evalúa al final del ciclo.

Sintaxis

```
do {
    <instrucciones>;
} while( <condición> );
```

Ejemplo:

```
1  import std.stdio;
2  int main () {
3      int a = 10;
4      do{
5          writeln("Valor de a: %d", a);
```

```

6      a = a + 1;
7  }while ( a < 20 );
8  return 0;
9  }

```

Listing 67. Ejemplo de estructura *do while* en D.

D soporta los controles de instrucciones *break* y *continue*. *break*: para terminar un ciclo o un instrucción de un *switch* y transferir la ejecución a la instrucción inmediata después del ciclo o del *switch*.

Sintaxis

break;

Ejemplo:

```

1  import std.stdio;
2  int main () {
3      int a = 10;
4      while ( a < 20 ) {
5          writeln("valor de a: %d", a);
6          a++;
7          if ( a > 15 ) {
8              break;
9          }
10     }
11     return 0;
12 }

```

Listing 68. Ejemplo de *break* en D.

continue: Forza la terminación del cuerpo de un ciclo y pasa el control, en el caso del *for* evalúa la condición y realiza el incremento; en el caso del *while* y *do...while* pasa el control para evaluar la condicional.

Sintaxis

continue;

Ejemplo:

```
1 import std.stdio;
2 int main () {
3     int a = 10;
4     do {
5         if ( a == 15 ) {
6             a = a + 1;
7             continue;
8         }
9         writeln("valor de a: %d", a);
10        a++;
11    } while ( a < 20 );
12    return 0;
13 }
```

Listing 69. Ejemplo de *continue* en D.

8.3.7 Enumeraciones

Una enumeración es usada para valores de nombres de constantes. Un tipo de enumeración es declarada usando la palabra clave *enum*.

Sintaxis

```
enum <nombre_enum> {
    <lista de enumeración>
}
```

Donde, *nombre_enum* especifica el nombre de tipo enumeración, *lista de enumeración* es la lista de identificadores separados por coma. Cada símbolo de la lista enumeración representa un valor entero. Por omisión, el valor del primer símbolo de la lista es 0.

Ejemplo:

```
1 import std.stdio;
2 enum Days { sun, mon, tue, wed, thu, fri, sat };
3 int main(string[] args)
4 {
```



```
5     Days day;
6     day = Days.mon;
7     writefln("Current Day: %d", day);
8     writefln("Friday : %d", Days.fri);
9     return 0;
10 }
```

Listing 70. Ejemplo de enumeración en D.

Propiedades:

- *init*: inicializa el primer valor en la enumeración.
- *min*: regresa el valor más pequeño de la enumeración.
- *max*: regresa el valor más grande de la enumeración.
- *sizeof*: retorna el tamaño de la enumeración.

Ejemplo:

```
1 import std.stdio;
2 // Inicializado con valor 1
3 enum Days { sun =1, m on, tue, wed, thu, fri, sat };
4 int main(string[] args){
5     writefln("Min : %d", Days.min);
6     writefln("Max : %d", Days.max);
7     writefln("Size of: %d", Days.sizeof);
8     return 0;
9 }
```

Listing 71. Ejemplo 2 de enumeración en D.

Ejemplo de enumeración anónima:

```
1 import std.stdio;
2 // Initialized sun with value 1
3 enum { sun , m on, tue, wed, thu, fri, sat };
4 int main(string[] args){
5     writefln("Sunday : %d", sun);
6     writefln("Monday : %d", m on);
7     return 0;
8 }
```

Listing 72. Ejemplo de enumeración anónima en D.

8.3.8 Módulos

El nombre de un modulo es el mismo que su nombre del archivo sin la extensión .d. Cuando explícitamente se especifica, el nombre del modulo es definido por la palabra clave *module* la cual debe aparecer como la primera linea como se muestra a continuación:

```
1  module empleado;
2      class Empleado {
3          // La definición de la clase va aquí.
4      }
```

La línea de module es opcional, cuando no se especifica es el mismo que el nombre del archivo sin la extensión .d.

D soporta Unicode en nombres de código fuente y del módulo. Sin embargo, el soporte Unicode de los sistemas de archivos puede variar. Por ejemplo, aunque la mayoría de los sistemas de archivos de Linux soportan Unicode, los nombres de los archivos en los sistemas de archivos de Windows no pueden distinguir entre letras mayúsculas y minúsculas. Además, la mayoría de los sistemas de archivos limitan los caracteres que se pueden utilizar en los nombres de archivos y directorios. Por razones de portabilidad, es recomendable utilizar solo letras ASCII minúsculas en los nombres de archivo.

Una combinación de módulos relacionados se llama paquete. Los archivos fuente que están dentro del mismo directorio se consideran pertenecientes al mismo paquete. El nombre del directorio se convierte en el nombre del paquete, que también debe ser especificado como la primera parte de los nombre del módulo.

Por ejemplo, si *empleado.d* y *oficina.d* están dentro del directorio *compañia*, el paquete se define para *empleado* y *oficina*.

```
1  module compañía.empleado;
2
3  class Empleado {
4  }
5
6  module compañía.oficina;
7
8  class Oficina {
9  }
```

La palabra clave de *import*, es para la introducción de un módulo en el módulo actual:

```
import std.stdio;
```

El nombre del módulo puede contener el nombre del paquete también. Por ejemplo, el *std.* indica que *stdio* es un módulo del paquete *std.*

8.3.9 Entrada y Salida básica en D

La salida básica a consola en D se hace con `write` y `writeln`

```
write("Dato ", n, ": ", foo, "\n");

write("Salida sin salto de línea");
```

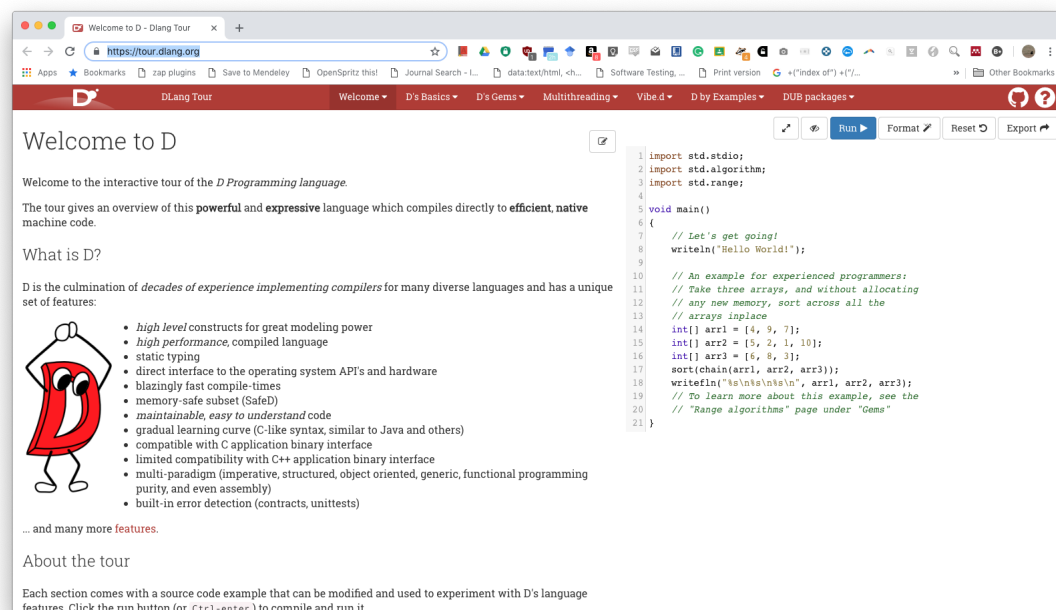
La lectura por otro lado podemos hacerla con la función `readln()` que regresa la entrada estándar a un arreglo de caracteres.

```
1 char[] nom;
2 nom = readln();
3 writef("Hola ", nom);
```

O se puede usar la lectura formateada que ofrece la función `readf()`. Las funciones se encuentran en la biblioteca de entrada y salida *std.stdio*.

8.3.10 Prueba D

Una especie de tutorial interactivo puede encontrarse en : <https://tour.dlang.org/>



Parte II

Programación Orientada a Objetos

Capítulo 9

Introducción a D

9.1 Introducción

Es un nuevo lenguaje de propósito general altamente influenciado por C++. Sin embargo, D es de más alto nivel pero se dice que conserva un rendimiento similar a C++ y ofrece la productividad de lenguajes como Ruby y Python. Creado por **Walter Bright** para la compañía **Digital Mars**.

A diferencia de la nueva ola de lenguajes, D no es un lenguaje tipo *script* ni incluye una máquina virtual para su ejecución. Sigue permitiendo acceso directo a las APIs del sistema operativo y al hardware.

D se puede considerar una reingeniería de C++ e influenciado por lenguajes como Java, Eiffel. D es un lenguaje de programación multiparadigma. Incluye paradigmas: Imperativo, Orientado a objetos, Funcional y Concurrente.

Actividad: lectura de artículo científico

http://www.computerworld.com.au/article/253741/-z_programming_languages_d

D es uno de los lenguajes usados por Facebook a partir de 2013, en sustitución de C++¹.

D, a diferencia de C++, utiliza una sintaxis y unas construcciones mucho más sencillas y lógicas. El rendimiento de C++, uno de sus puntos fuertes, también se ve reflejado en D y en algunas ocasiones incluso lo supera.

Características de D:

Gestión automática de memoria(recolección de basura): esto quiere decir que el programador seguirá creando los nuevos objetos con *new* pero ya no tendrá que preocuparse de borrarlos con *delete* porque existirá un **recolector de basura** que se encargará de eliminar automáticamente los objetos para los que ya no exista ninguna referencia. Sin embargo, si el programador lo desea, la recolección de basura

¹<http://forum.dlang.org/thread/l37h5s2gd81@digitalmars.com>

puede ser controlada: los programadores pueden agregar y excluir rangos de memoria de ser observados por el recolector, pueden pausar y reanudar el recolector y forzar un ciclo generacional o de recolección completa.

- **Gestión de errores mediante manejo de excepciones:** el sistema de manejo de excepciones es superior al de C++ al incorporar algunas características de lenguajes más recientes. Cuando una excepción producida en código D no se captura, se muestra un mensaje de error con información de la excepción.
- **Guardias de ámbito para asegurar la ejecución de código a la salida de un ámbito:** hay recursos como los archivos, los cerrojos y *mutex* que se siguen teniendo que liberar manualmente; pero hay otros. Realmente, usando las guardias de ámbito, la instrucción *finally* nunca es necesaria aunque se sigue manteniendo en el lenguaje. Las guardias de inclusión no sustituyen a la instrucción *except*; para capturar excepciones y realizar operaciones en la captura debemos seguir usándola.
- **Estructuración del código en módulos y paquetes:** la estructuración del código y las bibliotecas se hace usando módulos y paquetes. Un módulo no es más que un archivo fuente de código D (generalmente con extensión .d). Al contrario de lo que sucede en otros lenguajes, cuando importamos los símbolos de un módulo no hace falta anteponer el nombre del módulo con un punto antes de llamar a un símbolo.
- **Compatibilidad de llamada con C:** Debido a la difusión del lenguaje, la mayoría de las APIs de sistemas operativos y bibliotecas de sistemas están escritas en C u ofrecen una interfaz para el mismo. D puede acceder a bibliotecas de C. Como los tipos de datos de D suelen tener una correspondencia muy directa con los de C este proceso suele ser bastante sencillo; sin embargo para conversiones de archivos de cabecera .h más complicados se dispone de la herramienta *htod* que realiza la conversión de tipos y sintáxis de forma automática, tomando como entrada un archivo .h de C y generando un archivo .d que podemos incluir en nuestros proyectos.
- **Delegados, funciones anidadas y funciones literales:** existe un tipo de dato llamado **delegado** que puede usarse para pasar referencias a un método de una clase como parámetros para otras funciones y métodos. En concepto son similares a los punteros a método de C++ pero con una sintáxis tanto de declaración como de creación y uso mucho más sencilla. D permite tener funciones anidadas y funciones anónimas. Las funciones anidadas son las que están definidas dentro de otra función y son muy útiles para estructurar nuestro código de una forma más jerárquica. Las funciones anónimas son funciones sin nombre que suelen utilizarse como argumento para una función que espera recibir una función como argumento.

- **Declaración anticipada de funciones innecesaria:** para que una función pueda llamar a otra no es necesario que esta última haya sido declarada con anterioridad a la primera.

D retiene la habilidad de C++ de hacer código de bajo nivel, permitiendo incluir código en ensamblador.

9.2 Herramientas

Actualmente se encuentra accesible la versión 2.x, la cual fue presentada en junio de 2007. Existen diferentes implementaciones del lenguaje. La principal es desarrollada por la misma compañía (Digital Mars) en versiones Mac OSX Windows, Linux y FreeBSD. Otras distribuciones están disponibles, por ejemplo un compilador D.NET, obviamente para .NET

Si se usa la implementación de Digital Mars, entonces el compilador es *dmd*. La extensión usada para archivos de código fuente es *.d*. Existen otras extensiones relevantes (*.dd* para archivos de *Ddoc* – similar a *javadoc*-, *.di* para archivos de interfaces, y *.def* para archivos de definición de módulos, entre otros).

9.3 Fundamentos de D

9.3.1 Convenciones léxicas

Todos los archivos D tendrán extensión *.d*.

Ejemplo:

```
1 import std.stdio;
2 void main(string[] args) {
3     writeln("Hola, Mundo!");
4 }
```

Listing 73. Ejemplo "Hola, Mundo!".^{en D.}

Se importa la biblioteca *io* que ofrece las operaciones básicas de E/S. *writeln* que se utiliza en el programa anterior para la impresión de una línea de texto. El único módulo que utiliza este programa es *std.stdio*, que maneja la entrada y salida de datos.

Los comentarios en D pueden ser multilínea: el cual inicia con */** y cierra con **/*, y los comentarios de una sola línea con *//*

Los identificadores en D son los nombres que se les asigna para distinguir a las variables, funciones o cualquier otro elemento definido por el usuario. Un identi-

Secuencia de escape	Significado
\\	caracter \
\'	caracter '
\"	caracter "
\b	backspace
\f	Form feed
\n	Nueva línea
\r	Return
\t	Tabulador horizontal
\v	Tabulador vertical

ficador comienza con una letra A a la Z, o de la a a la z o un guión bajo (_)seguido de cero o mas letras, subrayado y los dígitos (0 a 9).

D no permite caracteres de puntuación tales como @, \$ y % dentro de los identificadores. D es un lenguaje sensible a mayúsculas y minúsculas.

9.3.2 Literales

Los tipos de valores constantes que forman parte del código fuente son llamados literales. Las literales pueden ser alguno de los tipos de datos básicos y pueden dividirse en Números Enteros, Números punto Flotante, Caracter, *Strings* y Valores Booleanos.

- Las literales enteros pueden ser: Decimal, Octal, Binario, Hexadecimal. Una literal entera también puede tener un sufijo que es una combinación de U o L, de *unsigned* y *Long*. Para escribir una literal hexadecimal integral se usa el prefijo 0x o 0X seguido por una secuencia de letras 0-9, a-f, A-F, o _.
- Literales de Punto Flotante: pueden ser especificados sistema decimal como 1.568 o en sistema hexadecimal. En el sistema decimal, un exponente puede ser representado agregando el carácter E o e seguido del valor del exponente, por ejemplo 2.3e4, aunque también 2.3e4 y 2.3e + 4 son lo mismo (A+ carácter pueden especificarse antes del valor del exponente). Por default la literal de punto flotante es *double*. F o f significa flotante.
- Literal Booleana: Existen dos valores posibles para las literal booleana (*True* y *False*).
- Literal Carácter: encerradas entre comillas simples(' '). Una literal de este tipo puede ser un carácter o una secuencia de escape('\t').

Ejemplo donde se utilizan las secuencias de escape:

```
1 import std.stdio;
2 int main(string[] args){
3     writeln("Hello\tWorld%c\n", '\x21');
4     writeln("Have a good day%c", '\x21');
5     return 0;
6 }
```

Listing 74. Ejemplo en D donde se utilizan las secuencias de escape.

- Literal *String*: son encerradas en comillas dobles, un *String* puede contener caracteres similares a las literales carácter (secuencias de escape, caracteres universales)
- Literal de Arreglo y asociación de arreglo: *Strings* son un particular tipo de arreglos. Una literal de arreglo es representada como una secuencia de valores separada por una coma encerrada por corchetes cuadrados. El tamaño del arreglo es la longitud de la lista separada por comas. El arreglo es no immutable, significa que puede modificarse después de su inicialización.
- Literal de Funciones: en algunos lenguajes, cada función tiene un nombre que se elige en el momento de la definición, subsecuentemente la función es llamada con ese nombre. Otros lenguajes tiene la posibilidad de definir funciones anónimas (funciones lambda). Esta característica vuelve poderoso al lenguaje al usar funciones de orden superior, estas funciones toman como parámetros y/o retornan otras funciones. D cuenta con literal de Funciones para definir funciones anónimas.
-

9.3.3 Variables

Una variable es sólo un nombre dado a un área de almacenamiento que nuestro programa puede manipular. Cada variable en D tiene un tipo específico, el cual determina el tamaño. Tipos básicos de variables en D: *char*, *int*, *float*, *doble*, *void*.

La siguiente tabla detalla los tamaños de almacenamiento de los tipos enteros (*int*):

La siguiente tabla detalla los tamaños de almacenamiento de los tipos de punto flotante:

La siguiente tabla detalla los tamaños de almacenamiento de los tipos carácter (*char*):

El tipo *void* se utiliza en dos situaciones:

Tipo	Tamaño	Rango de valor
bool	1 byte	false o true
byte	1 byte	-128 a 127
ubyte	1 byte	0 a 255
int	4 bytes	-2,147,483,648 a 2,147,483,647
uint	4 bytes	0 a 4,294,967,295
short	2 bytes	-32,768 to 32,767
ushort	2 bytes	0 a 65,535
long	8 bytes	-9223372036854775808 a 9223372036854775807
ulong	8 bytes	0 a 18446744073709551615

Tipo	Tamaño	Rango de valor
float	4 bytes	1.17549e-38 a 3.40282e+38
double	8 bytes	2.22507e-308 a 1.79769e+308
real	10 bytes	3.3621e-4932 a 1.18973e+4932
ifloat	4 bytes	1.17549e-38i a 3.40282e+38i
idouble	8 bytes	2.22507e-308i a 1.79769e+308i
ireal	10 bytes	3.3621e-4932 a 1.18973e+4932
cfloat	8 bytes	1.17549e-38+1.17549e-38i a 3.40282e+38+3.40282e+38i
cdouble	16 bytes	2.22507e-308+2.22507e-308i a 1.79769e+308+1.79769e+308i
creal	20 bytes	3.3621e-4932+3.3621e-4932i a 1.18973e+4932+1.18973e+4932i

Tipo	Tamaño
char	1 byte
wchar	2 bytes
dchar	4 bytes

1. La función devuelve void: funciones que no devuelven nada o que devuelven nulo. Una función sin valor de retorno tiene el tipo de retorno void.
2. Argumentos de la función vacío: Una función que no acepta parámetros, vacío, con ningún parámetro.

Definición de variables en D:

```
1 int i, j, k;  
2 char c, ch;  
3 float f, salary;  
4 double d;
```

Las variables también pueden ser inicializadas desde su declaración:

```
1 extern int d = 3, f = 5;  
2 int d = 3, f = 5;  
3 byte z = 22;  
4 char x = 'x';
```

En este [ejemplo](#) las variables han sido definidas en el tope del programa pero se redefinen e inicializan nuevamente dentro del main:

```
1 import std.stdio;  
2     int a = 10, b = 10;  
3     int c;  
4     float f;  
5     int main () {  
6         writeln("Value of a is : ", a);  
7         /* variable re definition: */  
8         int a, b;  
9         int c;  
10        float f;  
11        /* Initialization */  
12        a = 30;  
13        b = 40;  
14        writeln("Value of a is : ", a);  
15        c = a + b;  
16        writeln("Value of c is : ", c);  
17        f = 70.0/3.0;
```

Tipo	Operadores
Operadores Aritméticos:	+, -, *, /, %, ++, --
Operadores Relacionales:	==, !=, >, <, >=, <=
Operadores Lógicos:	&&, , !
Operadores Bit a bit:	&, y ^
Operadores de Asignación:	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, =
Operadores Misc:	sizeof, &(dirección de una variable), * (apuntador), ? :

```

18         writeln("Value of f is : ", f);
19         return 0;
20     }

```

Listing 75. Ejemplo de alcance en la definición de variables en D.
Con la siguiente salida:

```

Value of a is : 10
Value of a is : 30
Value of c is : 70
Value of f is : 23.3333

```

9.3.4 Operadores

Un operador es un símbolo que llama al compilador para ejecutar una operación específica, lógica o matemática. D provee los siguientes tipos de operadores:

Precedencia de operadores

Ver Cuadro 9.1

9.3.5 Arreglos

Los arreglos en D pueden ser estáticos o dinámicos.

Un arreglo estático se declara de la siguiente forma:

Sintaxis
<tipo> [<tamaño>] <variable>

De tal forma que un arreglo de 10 enteros quedaría de la siguiente forma:

```
int [10] arr
```

Cuadro 9.1. Precedencia de operadores en D

Categoría	Operador
Postfijo	[] -> . ++ - -
Unario	+ ! ~ ++ __ type* & sizeof
Multiplicativo	* / %
Aditivo	+ -
Shift	<< >>
Relacional	<= >=
Igualdad	== !=
Lógicos	AND &&
Lógicos	OR
Condicional	?:
Asignación	= += -= *= /= %= >>= <<= &= ^= =
Coma	,

Un arreglo dinámico por su parte se declararía de manera similar pero sin indicar el tamaño específico.

Sintaxis
<tipo> [] <variable>

Por lo que la declaración de un arreglo dinámico entero sería de la siguiente forma:

```
1 int [] a;
```

Los arreglos son objetos por lo que podemos obtener el número de elementos de un arreglo del atributo *length*.

Ejemplo:

```
1 int[10] a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
2 int[] b1, b2, b3, b4;
3
4 b1 = a;
5 b2 = a[];
6 b3 = a[0 .. a.length];
7 b4 = a[0 .. $];
8 writeln(b1);
9 writeln(b2);
```

```
10 writeln(b3);
11 writeln(b4);
12
```

La longitud de un arreglo dinámico se puede hacer asignando un nuevo valor al atributo *length*:

```
1 int [] a;
2
3 a.length=10;
4
```

9.3.6 Estructuras de control

if: una sentencia consiste de una expresión booleana seguida de una o mas declaraciones.

Sintaxis

```
if (<expresión_booleana>){
    /* se ejecuta si la expresión booleana es verdadera */
}
```

Ejemplo:

```
1 import std.stdio;
2 int main (){
3     /* Definición de variable local */
4     int a = 10;
5
6     if( a < 20 ){
7         writeln("a es menor que 20" );
8     }
9     writeln("valor de a es : %d", a);
10    return 0;
11 }
```


Listing 76. Ejemplo de estructura de control *if* en D.

if...else: una sentencia *if* puede estar seguida de una sentencia *else* opcional, que se ejecuta cuando la expresión booleana es falsa.

Sintaxis

```
if (<expresión_booleana>){  
  
} else {  
  
}
```

Ejemplo:

```
1      import std.stdio;  
2      int main (){  
3          int a = 100;  
4  
5          if( a < 20 ){  
6              writeln("a es menor que 20" );  
7          }  
8          else{  
9              writeln("a no es menor que 20" );  
10         }  
11         writeln("el valor de a es : %d", a);  
12         return 0;  
13     }
```

Listing 77. Ejemplo de estructura *if.. else* en D.

Switch: Permite a una variable ser evaluada con una lista de valores.

Sintaxis

```
switch(<expresión>){  
  case <expresión-constante> :  
    <instrucciones>;  
    break; /* opcional */  
  case <expresión-constante> :  
    <instrucciones>;  
    break; /* opcional */  
  
  default : /* Opcional */  
    <instrucciones>;  
}
```

```
1  import std.stdio;  
2  
3  int main (){  
4      char grade = 'B';  
5  
6      switch(grade){  
7          case 'A' :  
8              writeln("Excelente!" );  
9              break;  
10         case 'B' :  
11         case 'C' :  
12             writeln("Bien hecho" );  
13             break;  
14         case 'D' :  
15             writeln("Pasaste" );  
16             break;  
17         case 'F' :  
18             writeln("Trata de nuevo" );  
19             break;  
20         default :  
21             writeln("Reprobado" );  
22     }  
23     writeln("Calificación %c", grade );  
24 }
```

```

25     return 0;
26 }

```

Listing 78. Ejemplo de estructura *switch* en D.

while: repite una(s) instrucció(n)es mientras la condición es verdadera. Comprueba la condición antes de el cuerpo del ciclo.

Sintaxis

```

while(<condición>) {
    <instrucciones>;
}

```

Ejemplo:

```

1  import std.stdio;
2  int main ()      {
3      int a = 10;
4
5      while( a < 20 ){
6          writeln("Valor de a: %d", a);
7          a++;
8      }
9      return 0;
10 }
11

```

Listing 79. Ejemplo de estructura *while* en D.

for: ejecuta una secuencia de instrucciones múltiples veces.

Sintaxis

```

for ( <ini>; <condición>; <incremento> ){
    <instrucciones>;
}

```

1. `ini` se ejecuta primero y solo una vez.
2. La condición es evaluada. Si es verdadera, el cuerpo del ciclo se ejecuta. Si es falsa el cuerpo del ciclo no se ejecuta y cede el control a la siguiente instrucción después del ciclo.
3. Después de ejecutar el ciclo, se ejecuta la instrucción incremento.
4. La condición es evaluada nuevamente.

Ejemplo:

```
1  import std.stdio;
2  int main () {
3      for( int a = 10; a < 20; a = a + 1 ) {
4          writeln("Valor de a: %d", a);
5      }
6      return 0;
7  }
```

Listing 80. Ejemplo de estructura *for* en D.

do while: similar a *while* con la diferencia que la condición se evalúa al final del ciclo.

Sintaxis

```
do {
    <instrucciones>;
} while( <condición> );
```

Ejemplo:

```
1  import std.stdio;
2  int main () {
3      int a = 10;
4      do {
5          writeln("Valor de a: %d", a);
```

```

6      a = a + 1;
7  }while ( a < 20 );
8  return 0;
9  }

```

Listing 81. Ejemplo de estructura *do while* en D.

D soporta los controles de instrucciones *break* y *continue*. *break*: para terminar un ciclo o un instrucción de un *switch* y transferir la ejecución a la instrucción inmediata después del ciclo o del *switch*.

Sintaxis
<code>break;</code>

Ejemplo:

```

1  import std.stdio;
2  int main () {
3      int a = 10;
4      while ( a < 20 ) {
5          writeln("valor de a: %d", a);
6          a++;
7          if ( a > 15 ) {
8              break;
9          }
10     }
11     return 0;
12 }

```

Listing 82. Ejemplo de *break* en D.

continue: Forza la terminación del cuerpo de un ciclo y pasa el control, en el caso del *for* evalúa la condición y realiza el incremento; en el caso del *while* y *do...while* pasa el control para evaluar la condicional.

Sintaxis
<code>continue;</code>

Ejemplo:

```
1 import std.stdio;
2 int main () {
3     int a = 10;
4     do {
5         if ( a == 15 ) {
6             a = a + 1;
7             continue;
8         }
9         writeln("valor de a: %d", a);
10        a++;
11    } while ( a < 20 );
12    return 0;
13 }
```

Listing 83. Ejemplo de *continue* en D.

9.3.7 Enumeraciones

Una enumeración es usada para valores de nombres de constantes. Un tipo de enumeración es declarada usando la palabra clave *enum*.

Sintaxis

```
enum <nombre_enum> {
    <lista de enumeración>
}
```

Donde, *nombre_enum* especifica el nombre de tipo enumeración, *lista de enumeración* es la lista de identificadores separados por coma. Cada símbolo de la lista enumeración representa un valor entero. Por omisión, el valor del primer símbolo de la lista es 0.

Ejemplo:

```
1 import std.stdio;
2 enum Days { sun, mon, tue, wed, thu, fri, sat };
3 int main(string[] args)
4 {
```

```
5     Days day;
6     day = Days.mon;
7     writefln("Current Day: %d", day);
8     writefln("Friday : %d", Days.fri);
9     return 0;
10 }
```

Listing 84. Ejemplo de enumeración en D.

Propiedades:

- *init*: inicializa el primer valor en la enumeración.
- *min*: regresa el valor más pequeño de la enumeración.
- *max*: regresa el valor más grande de la enumeración.
- *sizeof*: retorna el tamaño de la enumeración.

Ejemplo:

```
1 import std.stdio;
2 // Inicializado con valor 1
3 enum Days { sun =1, m on, tue, wed, thu, fri, sat };
4 int main(string[] args){
5     writefln("Min : %d", Days.min);
6     writefln("Max : %d", Days.max);
7     writefln("Size of: %d", Days.sizeof);
8     return 0;
9 }
```

Listing 85. Ejemplo 2 de enumeración en D.

Ejemplo de enumeración anónima:

```
1 import std.stdio;
2 // Initialized sun with value 1
3 enum { sun , m on, tue, wed, thu, fri, sat };
4 int main(string[] args){
5     writefln("Sunday : %d", sun);
6     writefln("Monday : %d", m on);
7     return 0;
8 }
```

Listing 86. Ejemplo de enumeración anónima en D.

9.3.8 Módulos

El nombre de un modulo es el mismo que su nombre del archivo sin la extensión .d. Cuando explícitamente se especifica, el nombre del modulo es definido por la palabra clave *module* la cual debe aparecer como la primera linea como se muestra a continuación:

```
1  module empleado;
2      class Empleado {
3          // La definición de la clase va aquí.
4      }
```

La línea de module es opcional, cuando no se especifica es el mismo que el nombre del archivo sin la extensión .d.

D soporta Unicode en nombres de código fuente y del módulo. Sin embargo, el soporte Unicode de los sistemas de archivos puede variar. Por ejemplo, aunque la mayoría de los sistemas de archivos de Linux soportan Unicode, los nombres de los archivos en los sistemas de archivos de Windows no pueden distinguir entre letras mayúsculas y minúsculas. Además, la mayoría de los sistemas de archivos limitan los caracteres que se pueden utilizar en los nombres de archivos y directorios. Por razones de portabilidad, es recomendable utilizar solo letras ASCII minúsculas en los nombres de archivo.

Una combinación de módulos relacionados se llama paquete. Los archivos fuente que están dentro del mismo directorio se consideran pertenecientes al mismo paquete. El nombre del directorio se convierte en el nombre del paquete, que también debe ser especificado como la primera parte de los nombre del módulo.

Por ejemplo, si *empleado.d* y *oficina.d* están dentro del directorio *compañia*, el paquete se define para *empleado* y *oficina*.

```
1  module compañía.empleado;
2
3  class Empleado {
4  }
5
6  module compañía.oficina;
7
8  class Oficina {
9  }
```

La palabra clave de *import*, es para la introducción de un módulo en el módulo actual:


```
import std.stdio;
```

El nombre del módulo puede contener el nombre del paquete también. Por ejemplo, el *std.* indica que *stdio* es un módulo del paquete *std.*

9.3.9 Entrada y Salida básica en D

La salida básica a consola en D se hace con `write` y `writeln`

```
write("Dato ", n, ": ", foo, "\n");
write("Salida sin salto de línea");
```

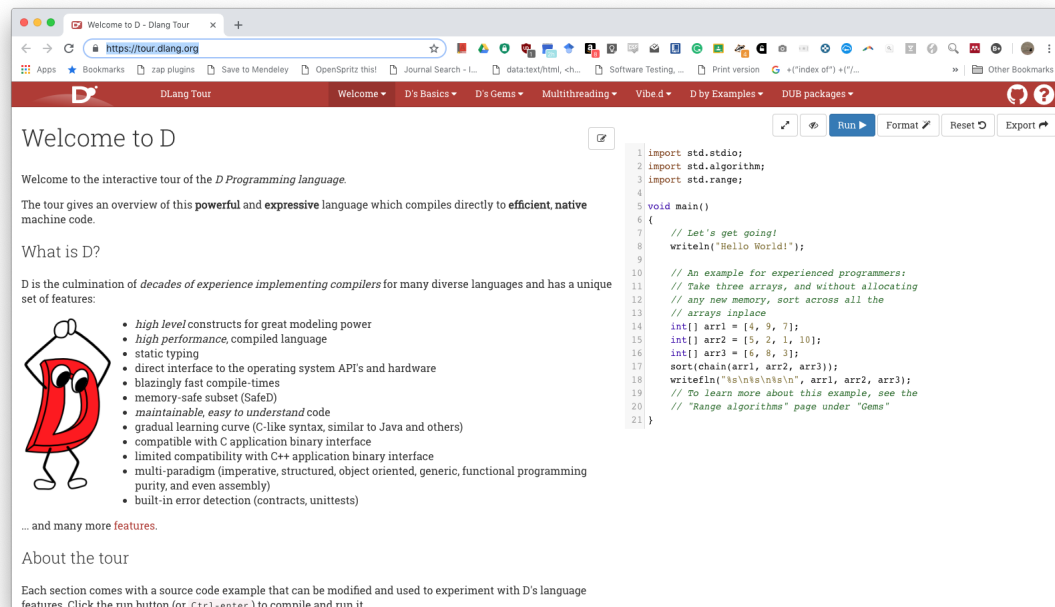
La lectura por otro lado podemos hacerla con la función `readln()` que regresa la entrada estándar a un arreglo de caracteres.

```
1 char[] nom;
2 nom = readln();
3 writef("Hola ", nom);
```

O se puede usar la lectura formateada que ofrece la función `readf()`. Las funciones se encuentran en la biblioteca de entrada y salida *std.stdio*.

9.3.10 Prueba D

Una especie de tutorial interactivo puede encontrarse en : <https://tour.dlang.org/>



Parte III

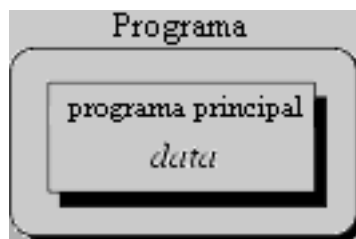
Programación Orientada a Objetos

Capítulo 10

Introducción a la programación orientada a objetos [? ?]

10.1 Programación no estructurada

Comúnmente, las personas empiezan a aprender a programar escribiendo programas pequeños y sencillos consistentes en un solo programa principal.



Aquí "programa principal" se refiere a una secuencia de comandos o instrucciones que modifican datos que son a su vez globales en el transcurso de todo el programa.

10.2 Programación procedural

Con la programación procedural se pueden combinar las secuencias de instrucciones repetitivas en un solo lugar.

Una llamada de procedimiento se utiliza para invocar al procedimiento.

Después de que la secuencia es procesada, el flujo de control procede exactamente después de la posición donde la llamada fue hecha.



Al introducir parámetros, así como procedimientos de procedimientos (subprocedimientos) los programas ahora pueden ser escritos en forma más estructurada y con menos errores.

Por ejemplo, si un procedimiento ya es correcto, cada vez que es usado produce resultados correctos. Por consecuencia, en caso de errores, se puede reducir la búsqueda a aquellos lugares que todavía no han sido revisados.

De este modo, un programa puede ser visto como una secuencia de llamadas a procedimientos. El programa principal es responsable de pasar los datos a las llamadas individuales, los datos son procesados por los procedimientos y, una vez que el programa ha terminado, los datos resultantes son presentados.

Así, el flujo de datos puede ser ilustrado como una gráfica jerárquica, un árbol, como se muestra en la figura para un programa sin sub-procedimientos.



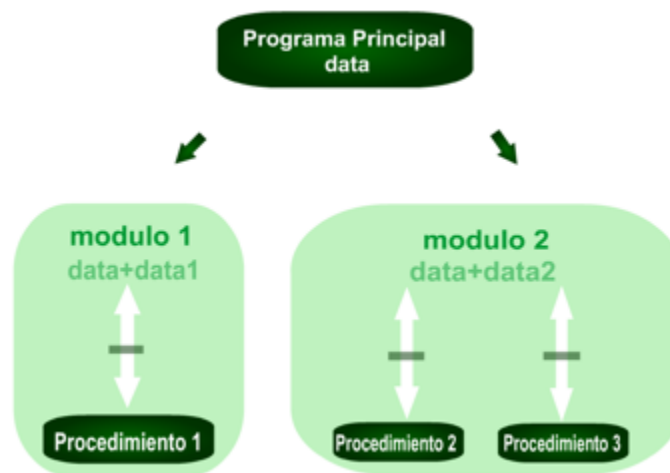
10.3 Programación modular

En la programación modular, los procedimientos con una funcionalidad común son agrupados en módulos separados.

Un programa por consiguiente, ya no consiste solamente de una sección. Ahora está dividido en varias secciones más pequeñas que interactúan a través de llamadas a procedimientos y que integran el programa en su totalidad.

Cada módulo puede contener sus propios datos. Esto permite que cada módulo maneje un estado interno que es modificado por las llamadas a procedimientos de ese módulo.

Sin embargo, solamente hay un estado por módulo y cada módulo existe cuando más una vez en todo el programa.



10.4 Datos y Operaciones separados

La separación de datos y operaciones conduce usualmente a una estructura basada en las operaciones en lugar de en los datos: **los Módulos agrupan las operaciones comunes en forma conjunta.**

Al programar entonces se usan estas operaciones proveyéndoles explícitamente los datos sobre los cuáles deben operar.

La estructura de módulo resultante está por lo tanto orientada a las operaciones más que sobre los datos. Se podría decir que las operaciones definidas especifican los datos que serán usados.

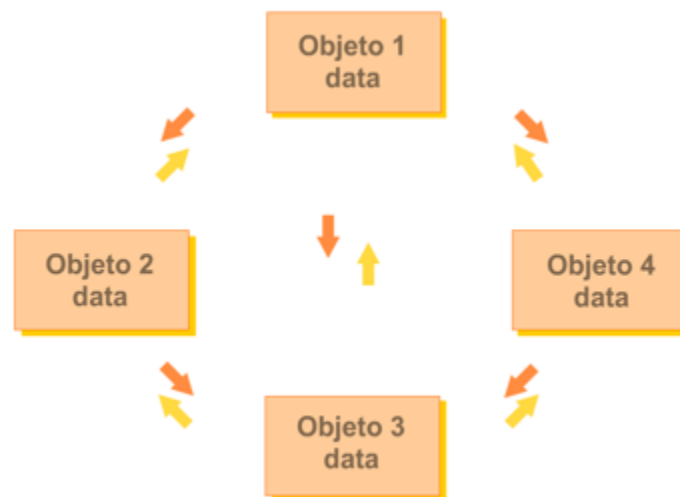
En la programación orientada a objetos, la estructura se organiza por los datos. Se escogen las representaciones de datos que mejor se ajusten a tus requerimientos. Por consecuencia, los programas se estructuran por los datos más que por las operaciones.

Los datos especifican las operaciones válidas. Ahora, los módulos agrupan representaciones de datos en forma conjunta.

10.5 Programación orientada a objetos

La programación orientada a objetos resuelve algunos de los problemas que se acaban de mencionar. De alguna forma se podría decir que obliga a prestar atención a los datos.

En contraste con las otras técnicas, ahora tenemos una telaraña de objetos interactuantes, cada uno de los cuáles manteniendo su propio estado.



Por ejemplo, en la programación orientada a objetos deberíamos tener tantos objetos de pila como sea necesario. En lugar de llamar un procedimiento al que le debemos proveer el manejador de la pila correcto, mandaríamos un mensaje directamente al objeto pila en cuestión.

En términos generales, cada objeto implementa su propio módulo, permitiendo por ejemplo que coexistan muchas pilas. Cada objeto es responsable de inicializarse y destruirse en forma correcta.

¿No es ésta solamente una manera más elegante de técnica de programación modular?

Podría ser, si esto fuera todo acerca de la orientación a objetos. De hecho se puede tratar de programar de esta forma sin POO. Pero eso no es todo lo que es la POO.

10.6 Tipos de Datos Abstractos

Algunos autores describen la programación orientada a objetos como programación de tipos de datos abstractos y sus relaciones. Los tipos de datos abstractos son como un concepto básico de orientación a objetos.

10.6.1 Los problemas

La primera cosa con la que uno se enfrenta cuando se escriben programas es el problema.

Típicamente, uno se enfrenta a problemas "de la vida real" y nos queremos facilitar la existencia por medio de un programa para manejar dichos problemas.

Sin embargo, los problemas de la vida real son nebulosos y la primera cosa que se tiene que hacer es tratar de entender el problema para separar los detalles esenciales de los no esenciales: tratando de obtener tu propia perspectiva abstracta, o modelo, del problema. Este proceso de modelado se llama abstracción y se ilustra en la Figura:



El modelo define una perspectiva abstracta del problema. Esto implica que el modelo se enfoca solamente en aspectos relacionados con éste y que uno trata de definir propiedades del mismo. Estas propiedades incluyen

- los datos que son afectados
- las operaciones que son identificadas

por el problema.

Para resumir, la abstracción es la estructuración de un problema nebuloso en entidades bien definidas por medio de la definición de sus datos y operaciones. Consecuentemente, estas entidades combinan datos y operaciones. No están desacoplados unos de otras.

10.6.2 Tipos de Datos Abstractos y Orientación a Objetos

Los TDAs permiten la creación de instancias con propiedades bien definidas y comportamiento bien definido. En orientación a objetos, nos referimos a los TDAs como clases. Por lo tanto, una clase define las propiedades de objetos en un ambiente orientado a objetos.

Los TDAs definen la funcionalidad al poner especial énfasis en los datos involucrados, su estructura, operaciones, así como en axiomas y precondiciones. Consecuentemente, la programación orientada a objetos es "programación con TDAs": al combinar la funcionalidad de distintos TDAs para resolver un problema. Por lo tanto, instancias (objetos) de TDAs (clases) son creadas dinámicamente, usadas y destruidas.

10.7 Conceptos de básicos de objetos

La programación tradicional separa los datos de las funciones, mientras que la programación orientada a objetos define un conjunto de objetos donde se combina de forma modular los datos con las funciones.

Aspectos principales:

1. Objetos.

- El objeto es la **entidad básica** del modelo orientado a objetos.
- El objeto integra una **estructura de datos** (atributos) y un **comportamiento** (operaciones).
- Se distinguen entre sí por medio de su propia identidad, aunque internamente los valores de sus atributos sean iguales.

2. Clasificación.

- Las clases **describen** posibles objetos, con una estructura y comportamiento común.
- Los objetos que contienen los mismos atributos y operaciones pertenecen a la misma clase.
- La estructura de clases **integra** las **operaciones** con los

atributos a los cuales se aplican.

3. Instanciación

- El proceso de **crear** objetos que pertenecen a una clase se denomina instanciación. (El objeto es la instancia de una clase).
- Pueden ser instanciados un número indefinido de objetos de cierta clase.

4. Generalización.

- En una jerarquía de clases, se **comparten** atributos y operaciones entre clases basados en la generalización de clases.
- La jerarquía de generalización se construye mediante la herencia.
- Las clases más generales se conocen como superclases. (clase padre)
- Las clases más especializadas se conocen como subclases. (clases hijas)
- La herencia puede ser simple o múltiple.

5. Abstracción.

- La abstracción se concentra en lo primordial de una entidad y no en sus propiedades secundarias.
- Además en lo que el objeto hace y no en cómo lo hace.
- Se da énfasis a cuales son los objetos y no cómo son usados.

6. Encapsulación.

- Encapsulación o encapsulamiento es la **separación** de las **propiedades externas** de un objeto de los **detalles de implementación** internos del objeto.
- Al separar la interfaz del objeto de su implementación, se limita la complejidad al mostrarse sólo la información relevante.
- Disminuye el impacto a cambios en la implementación, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa.

7. Modularidad

- El encapsulamiento de los objetos trae como consecuencia una gran modularidad.
- Cada módulo se concentra en una sola clase de objetos.
- Los módulos tienden a ser pequeños y concisos.
- La modularidad facilita encontrar y corregir problemas.
- La complejidad del sistema se reduce facilitando su mantenimiento.

8. Extensibilidad.

- La extensibilidad permite hacer cambios en el sistema sin afectar lo que ya existe.

- Nuevas clases pueden ser definidas sin tener que cambiar la interfaz del resto del sistema.
- La definición de los objetos existentes puede ser extendida sin necesidad de cambios más allá del propio objeto.

9. Polimorfismo (de subtipos).

- El polimorfismo es la característica de definir las **mismas operaciones** con **diferente comportamiento** en diferentes clases.
- Se permite llamar una operación sin preocuparse de cuál implementación es requerida en que clase, siendo responsabilidad de la jerarquía de clases y no del programador.

10. Reusabilidad de código.

- La orientación a objetos apoya el reuso de código en el sistema.
- Los componentes orientados a objetos se pueden utilizar para estructurar bibliotecas resuables.
- El reuso reduce el tamaño del sistema durante la creación y ejecución.
- Al corresponder varios objetos a una misma clase, se guardan los atributos y operaciones una sola vez por clase, y no por cada objeto.
- La herencia es uno de los factores más importantes contribuyendo al incremento en el reuso de código dentro de un proyecto.

Actividad: lectura de artículo científico

Wegner, Peter. "Classification in object-oriented systems." ACM Sigplan Notices 21.10 (1986): 173-182.

10.8 Lenguajes de programación orientada a objetos

Simula I (1967) fue originalmente diseñado para problemas de simulación y fue el primer lenguaje en el cual los datos y procedimientos estaban unificados en una sola entidad. Su sucesor **Simula** (1973), derivó definiciones formales a los conceptos de objetos y clase.

Simula sirvió de base a una generación de lenguajes de programación orientados a objetos. Es el caso de **C++** (1985), **Eiffel** (1986) y **Beta**. (1987)

Ada (1983), se derivan de conceptos similares, e incorporan el concepto de jerarquía de herencia. **CLU -clusters-** (1986) también incorpora herencia.

Cuadro 10.1. Características de LPOO

	Ada 95	Eiffel	Smalltalk	C++	Java
Paquetes	Sí	No	No	No	Sí
Herencia	Simple	Múltiple	Simple	Múltiple	Simple
Control de tipos	Fuerte	Fuerte	Sin tipos	Fuerte	Fuerte
Enlace	Dinámico	Dinámico	Dinámico	Dinámico	Dinámico
Concurrencia	Sí	No	No	No	Sí
Recolección de basura	No	Sí	Sí	No	Sí
Afirmaciones	No	Sí	No	No	Sí*
Persistencia	No	No	No	No	No
Generecidad	Sí	Sí	Sí	Sí	Sí*

*Afirmaciones y generecidad en Java fueron incluidos a partir de la versión 5 (1.5) del lenguaje.

Smalltalk es descendiente directo de **Simula**, generaliza el concepto de objeto como única entidad manipulada en los programas. Existen tres versiones principales: **Smalltalk-72**, introdujo el paso de mensajes para permitir la comunicación entre objetos. **Smalltalk-76** que introdujo herencia. **Smalltalk-80** se inspira en **Lisp**.

Lisp contribuyó de forma importante a la evolución de la programación orientada a objetos.

Flavors (1986) maneja herencia múltiple apoyada con facilidades para la combinación de métodos heredados.

CLOS (1989), es el estándar del sistema de objetos de **Common Lisp**.

Los programas de programación orientada a objetos pierden eficiencia ante los lenguajes imperativos, pues al ser interpretado estos en la arquitectura *von Neumann* resulta en un **excesivo** manejo dinámico de la memoria por la constante creación de objetos, así como una fuerte carga por la división en múltiples operaciones (métodos) y su ocupación. Sin embargo se gana mucho en **comprensión** de código y **modelado** de los problemas.

10.8.1 Características de los algunos LPOO¹

En el Cuadro 10.1 podemos ver algunas características de lenguajes de programación orientados a objetos.

¹Lenguajes de Programación Orientada a Objetos

Capítulo 11

Abstracción de datos: Clases y objetos

11.1 Clases

Se mencionaba anteriormente que la base de la programación orientada a objetos es la abstracción de los datos o los TDAs. La abstracción de los datos se da realmente a través de las clases y objetos.

Concepto
Def. Clase. Se puede decir que una clase es la implementación real de un TDA, proporcionando entonces la estructura de datos necesaria y sus operaciones. Los datos son llamados atributos y las operaciones se conocen como métodos [?].

La unión de los atributos y los métodos dan forma al **comportamiento** (comportamiento común) de un grupo de objetos. La clase es entonces como la definición de un esquema dentro del cual encajan un conjunto de objetos.

El comportamiento debe ser descrito en términos de **responsabilidades** [?]. Resolviendo el problema bajo esos términos permite una mayor independencia entre los objetos, al elevar el nivel de abstracción.

En Programación Estructurada el programa opera **sobre** estructuras de datos. En contraste en Programación Orientada a Objetos, el programa solicita a las estructuras de datos que ejecuten un servicio.

Ejemplos de clases:

- automóvil,
- persona,
- libro,

- revista,
- reloj,
- silla,
- ...

11.2 Objetos e instancias

Una de las características más importantes de los lenguajes orientados a objetos es la **instanciación**. Esta es la capacidad que tienen los nuevos tipos de datos, para nuestro caso en particular las clases de ser **instanciadas** en cualquier momento.

El instanciar una clase produce un objeto o instancia de la clase requerida. Todos los objetos son **instancia** de una clase[?].

Concepto
Def. Objeto. Un objeto es una instancia de una clase. Puede ser identificado en forma única por su nombre y define un estado, el cuál es representado por los valores de sus atributos en un momento en particular [?].

El estado de un objeto cambia de acuerdo a los métodos que le son aplicados. Nos referimos a esta posible secuencia de cambios de estado como el comportamiento del objeto:

Concepto
Def. Comportamiento. El comportamiento de un objeto es definido por un conjunto de métodos que le pueden ser aplicados [?].

11.2.1 Instanciación

Los objetos pueden ser creados de la misma forma que una estructura de datos:

1. **Estáticamente.** En **tiempo de compilación** se le asigna un área de memoria.
2. **Dinámicamente.** Se le asigna un área de memoria en **tiempo de ejecución** y su existencia es temporal. Es necesario liberar espacio cuando el objeto ya no es útil; para esto puede ser que el lenguaje proporcione mecanismos de recolección de basura.

En Java, los objetos sólo existen de manera dinámica, además de que incluye un recolector de basura para no dejar como responsabilidad del usuario la eliminación de los objetos de la memoria.

11.3 Clases en C++

Una clase entonces, permite encapsular la información a través de atributos y métodos que la utilizan, ocultando la misma y la implementación del comportamiento de las clases.

La definición de una clase define nuevos TDAs y la definición en C++ consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves y finalizando con “;”. Notar la similitud de las clases con las estructuras de C.

El cuerpo de la clase contiene la declaración de los atributos de la clase (variables) y la declaración de los métodos (funciones). Tanto los atributos como los métodos pertenecen exclusivamente a la clase y sólo pueden ser usados a través de un objeto de esa clase.

Sintaxis

```
class <nombre_clase> {  
    <cuerpo de la clase>  
};
```

Ejemplo:

```
1 class Ejemplo1 {  
2     int x;  
3     float y;  
4     void fun(int a, float b) {  
5         x=a;  
6         y=b;  
7     }  
8 };
```

11.4 Miembros de una clase en C++

Una clase está formada por un conjunto de miembros que pueden ser datos, funciones, clases anidadas, enumeraciones, tipos de dato, etc. (amigos) Por el momento nos vamos a centrar en los datos y las funciones (atributos y métodos).

Es importante señalar que un miembro no puede ser declarado más de una vez¹. Tampoco es posible añadir miembros después de la declaración de la clase.

Ejemplo:

```
1 class Ejemplo2{
2     int i;
3     int i;    //error
4     int j;
5     int func(int, int);
6 };
```

11.4.1 Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

11.4.2 Métodos miembro

Los métodos al igual que los atributos, deben ser definidos en la clase, pero el cuerpo de la función puede ir dentro o fuera de la clase. Si un método se declara completo dentro de la clase, se considera como inline.

La declaración dentro de la clase no cambia con respecto a la declaración de una función, salvo que se hace dentro de la clase. Veamos un ejemplo parecido al inicial de esta sección, pero ahora con el cuerpo de un método fuera del cuerpo de la clase.

Ejemplo:

```
1 //código en ejemplo3.h
2 class Ejemplo3 {
3     public:
4     int x;
5     float y;
6     int funX(int a) {
7         x=a;
8         return x;
9     }
10    float funY(float);
11 };
```

¹Aunque existe el concepto de sobrecarga que se verá más adelante

Podemos ver que en la definición de la clase se incluye un método en línea y un prototipo de otro método.

Para definir un método miembro de una clase fuera de la misma, se debe escribir antes del nombre del método, el nombre de la clase con la que el método está asociado. Para esto se ocupa el operador de resolución de alcance (o de ámbito) ::.

Continuación del ejemplo:

```
1 float Ejemplo3::funY(float b) {
2     y=b;
3     return y;
4 }
```

Reiteramos que al declarar los métodos fuera de la clase no puede mencionarse la declaración de un método que no esté contemplado dentro de la clase. Si esto fuera válido, cualquier método podría ganar acceso a la clase con sólo declarar una función adicional.

Ejemplo:

```
1 //error en declaración de un método
2 class x{
3     public:
4     int a;
5     f();
6 };
7
8 int x::g() {           //error, el metodo debe ser f()
9     return a*=3.1234;
10 }
```

La declaración de una función miembro es considerada dentro del ámbito de su clase. Lo cual significa que puede usar nombres de miembros de la clase directamente sin usar el operador de acceso de miembro de la clase.

Recordar que por convención en la programación orientada a objetos las funciones son llamadas métodos y la invocación o llamada se conoce como mensaje.

11.4.3 Un vistazo al acceso a miembros

Otra de las ventajas de la POO es la posibilidad de encapsular datos, ocultándolos de otros objetos si es necesario. Para esto existen principalmente dos calificadores que definen a los datos como **públicos** o **privados**.

Miembros públicos

Se utiliza cuando queremos dar a usuarios de una clase (e.g., otras clases) el acceso a miembros de esa clase, los miembros deben ser declarados públicos.

Sintaxis
<pre>public: <definición de miembros></pre>

Miembros privados

Si queremos ocultar ciertos miembros de una clase de los usuarios de la misma, debemos declarar a los miembros como privados. De esta forma nadie más que los miembros de la clase pueden usar a los miembros privados. Con excepción de las funciones amigas. Por omisión los miembros se consideran privados. En una estructura se consideran públicos por omisión.

Sintaxis
<pre>private: <definición de miembros></pre>

Normalmente, los atributos de la clase deben ser privados; así como los métodos que no sean necesarios externamente o que puedan conducir a un estado inconsistente del objeto².

En el caso de los atributos, estos al ser privados deberían de contar con métodos de modificación y de consulta pudiendo incluir alguna validación.

Es una buena costumbre de programación acceder a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo.

Ejemplo:

²Un estado inconsistente sería ocasionado por una modificación indebida de los datos, por ejemplo una modificación sin validación.

```
1 //código en ejemplo3.h
2 class Fecha {
3     private:
4         int dia;
5         int mes;
6         int an;
7
8     public:
9         bool setDia(int);    //poner día
10        int getDia();        //devuelve día
11        bool setMes(int);
12        int getMes();
13        bool setAn(int);
14        int getAn();
15 };
```

11.5 Objetos de clase en C++

Ya se ha visto como definir una clase, declarando sus atributos y sus operaciones, mismas que pueden ir dentro de la definición de la clase (*inline*) o fuera. Ahora vamos a ver como es posible crear objetos o instancias de esa clase.

Hay que recordar que una de las características de los objetos es que cada uno guarda un estado particular de acuerdo al valor de sus atributos³.

Lo más importante de los lenguajes orientados a objetos es precisamente el objeto, el cual es una identidad lógica que contiene datos y código que manipula esos datos. En C++, un objeto es una variable de un tipo definido por el usuario[?]. Un [ejemplo](#) completo:

```
1 #include <iostream>
2 using namespace std;
3
4 class Ejemplo3 {
5     public:
6         int i;
7         int j;
8 };
9
10 int main() {
```

³A diferencia de la programación modular, donde cada módulo tiene un solo estado.

```
11     Ejemplo3 e1;
12     Ejemplo3 e2;
13     e1.i=10;
14     e1.j=20;
15
16     e2.i=100;
17     e2.j=20;
18     cout<<e1.i<<endl;
19     cout<<e2.i<<endl;
20
21     return 0;
22 }
```

Listing 87. Ejemplo de clases en C++.

Otro [ejemplo](#), una cola:

```
1  class Cola{
2      private:
3          int q[10];
4          int sloc, rloc;
5
6      public:
7          void ini() { //función en línea
8              sloc=rloc=-1;
9          }
10         bool set(int);
11         int get();
12 };
13
14 #include <iostream>
15 #include "Cola.h"
16
17 using namespace std;
18
19 bool Cola::set(int val){
20     if(sloc>=10){
21         cout<<"la cola esta llena";
22         return false;
23     }
24     sloc++;
25     q[sloc]=val;
```

```
26         return true;
27     }
28     int Cola::get () {
29         if(rloc==sloc)
30             cout<<"la cola esta vacia";
31         else {
32             rloc++;
33             return q[rloc];
34         }
35     }
36
37
38     //cola definida en un arreglo
39     #include <iostream>
40     #include "Cola.h"
41
42     using namespace std;
43
44     int main() {
45         Cola a,b, *pCola= new Cola;  // *pCola=nullptr y después asignarle
46
47         a.ini();
48         b.ini();
49         pCola->ini();
50         a.set(1);
51         b.set(2);
52         pCola->set(3);
53         a.set(11);
54         b.set(22);
55         pCola->set(33);
56         cout<<a.get()<<endl;
57         cout<<a.get()<<endl;
58         cout<<b.get()<<endl;
59         cout<<b.get()<<endl;
60         cout<<pCola->get()<<endl;
61         cout<<pCola->get()<<endl;
62
63         delete pCola;
64         return 0;
65     }
```

Listing 88. Ejemplo 2 de clases en C++, una estructura de cola simple.

Nota

Tomar en cuenta las instrucciones siguientes para el precompilador en el manejo de múltiples archivos.

```
1  #ifndef CCOLA_H
2  #define CCOLA_H
3  <definición de la clase>
4  #endif
```

11.6 Clases en Java

La definición en **Java**, de manera similar a C++, consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves.

Sintaxis

```
class <nombre_clase> {  
    <cuerpo de la clase>  
}
```

Ejemplo⁴:

```
1 public class Ejemplo1 {  
2     int x;  
3     float y;  
4     void fun(int a, float b) {  
5         x=a;  
6         y=b;  
7     }  
8 }
```

11.7 Miembros de una clase en Java

Los miembros en Java son esencialmente los atributos y los métodos de la clase.
Ejemplo:

```
1 class Ejemplo2{  
2     int i;  
3     int i;    //error  
4     int j;  
5     int func(int, int) {}  
6 }
```

⁴Algunos ejemplos como este no son programas completos, sino simples ejemplos de clases. Podrán ser compilados pero no ejecutados directamente. Para que un programa corra debe contener o ser una clase derivada de *applet*, o tener un método *main*.

11.7.1 Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

La sintaxis mínima es la siguiente:

Sintaxis

```
tipo nombreAtributo;
```

Los atributos pueden ser inicializados desde su lugar de declaración:

```
tipo nombreAtributo = valor;
```

o, en el caso de variables de objetos:

```
tipo nombreAtributo = new Clase();
```

11.7.2 Métodos miembro

Un método es una operación que pertenece a una clase. No es posible declarar métodos fuera de la clase. Además, en Java no existe el concepto de método prototipo como en C++.

Sin embargo, igual que en C++, la declaración de una función ó método miembro es considerada dentro del ámbito de su clase.

La sintaxis básica para declarar a un método:

Sintaxis

```
tipoRetorno nombreMétodo ( [<parámetros>] ) {  
    <instrucciones>  
}
```

Un aspecto importante a considerar es que el paso de parámetros en Java es realizado exclusivamente por valor. Datos básicos y objetos son pasados por valor. Pero los objetos no son pasados realmente, se pasan las referencias a los objetos (i.e., una copia de la referencia al objeto).

11.7.3 Un vistazo al acceso a miembros

Si bien en Java existen también los miembros públicos y privados, estos tienen una sintaxis diferente a C++. En Java se define el acceso a cada miembro de manera unitaria, al contrario de la definición de acceso por grupos de miembros de C++.

Miembros públicos

Sintaxis
<code>public</code> <definición de miembro>

Miembros privados

Sintaxis
<code>private</code> <definición de miembro>

Si se omite el nivel de acceso de un miembro, se considera como *acceso de paquete*. Es decir, se tiene acceso al miembro únicamente dentro del paquete en el que la clase esta declarada⁵.

Recordatorio
Es una buena costumbre de programación acceder a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo. Estos métodos de acceso y modificación comúnmente tienen el prefijo <i>get</i> y <i>set</i> , respectivamente.

Ejemplo:

```

1 class Fecha {
2     private int dia;
3     private int mes, an;
4
5     public boolean setDia(int d) {} //poner día
6     public int getDia() {} //devuelve día
7     public boolean setMes(int m) {}
8     public int getMes() {}
9     public boolean setAño(int a) {}
10    public int getAño() {}
11 }
```

⁵Fuente: [Java documentation](#)

11.8 Objetos de clase en Java

En Java todos los objetos son creados dinámicamente, por lo que se necesita reservar la memoria de estos en el momento en que se van a ocupar. El operador de Java está basado también en el de C++ y es *new*⁶.

11.8.1 Asignación de memoria al objeto

El operador *new* crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. Esta referencia debe de recibirla un identificador de la misma clase de la que se haya reservado la memoria.

Sintaxis

```
<identificador> = new Clase();
```

o en el momento de declarar a la variable de objeto:

```
Clase <identificador> = new Clase();
```

El concepto de *new* va asociado de la noción de constructor, pero esta se verá más adelante, por el momento basta con adoptar esta sintaxis para poder completar ejemplos de instanciación.

Un [ejemplo](#) completo:

```
1 public class Ejemplo3 {  
2     public int i, j;  
3  
4     public static void main(String argv[]) {  
5         Ejemplo3 e3= new Ejemplo3();  
6         Ejemplo3 e1= new Ejemplo3();  
7  
8         e1.i=10;  
9         e1.j=20;  
10        e3.i=100;  
11        e3.j=20;
```

⁶La instrucción *new*, ya había sido usada para reservar memoria a un arreglo, ya que estos son considerados objetos.

```
12         System.out.println(e1.i);
13         System.out.println(e3.i);
14     }
15 }
```

Listing 89. Ejemplo de clase en Java.

Otro [ejemplo](#), una estructura de cola:

```
1  class Cola{
2      private int q[];
3      private int sloc, rloc;
4
5      public void ini() {
6          sloc=rloc=-1;
7      }
8
9
10     public boolean set(int val){
11         if(sloc>=10){
12             System.out.println("la cola esta llena");
13             return false;
14         }
15         sloc++;
16         q[sloc]=val;
17         return true;
18     }
19
20     public int get(){
21         if(rloc==sloc){
22             System.out.println("la cola esta vacia");
23             return -1;
24         }
25         else {
26             rloc++;
27             return q[rloc];
28         }
29     }
30 }
31
32 public class PruebaCola {
33     public static void main(String argv[]){
```

```
34
35         Cola a= new Cola(); // new crea realmente el objeto
36         Cola b= new Cola(); // reservando la memoria
37         Cola pCola= new Cola();
38
39         //Inicializacion de los objetos
40         a.ini();
41         b.ini();
42         pCola.ini();
43
44         a.set(1);
45         b.set(2);
46         pCola.set(3);
47         a.set(11);
48         b.set(22);
49         pCola.set(33);
50
51         System.out.println(a.get());
52         System.out.println(a.get());
53         System.out.println(b.get());
54         System.out.println(b.get());
55         System.out.println(pCola.get());
56         System.out.println(pCola.get());
57     }
58 }
```

Listing 90. Ejemplo de clase con una estructura de Cola simple en Java.

11.9 Clases en Ruby

La definición en **Ruby**, de manera similar a Java y C++, consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase[?].

Sintaxis
<pre>class <nombre_clase> <cuerpo de la clase> end</pre>

Si la clase es contenida en un archivo la convención es que el nombre del archivo coincida con el nombre de la clase pero iniciando en minúsculas. Una clase *Prueba* estará idealmente en un archivo *prueba.rb* (en Windows se usa *.rbw*)

Ejemplo⁷:

```
1 class Ejemplo01
2     def fun a,b
3         x=a
4         y=b
5     end
6 end
```

11.10 Miembros de una clase en Ruby

Los miembros en Ruby son esencialmente los atributos y los métodos de la clase.

11.10.1 Métodos miembro

Un método es una operación que pertenece a una clase. Los métodos son muy importantes en Ruby. En Ruby no existen funciones, pues todo el código es representado en métodos (asociados a objetos). Aunque Ruby permite definir código

⁷Algunos ejemplos como este no son programas completos, sino simples ejemplos de clases. Ruby es un lenguaje interpretado. No existe un método principal que inicie la ejecución. El intérprete recibe una lista de instrucciones y éste comienza ejecutando de la línea inicial hasta la última línea.

similar al concepto de funciones en otros lenguajes, en realidad ese código es un método asociado (por omisión) a un objeto.

La sintaxis básica para declarar a un método:

Sintaxis
<pre>def <nombreMétodo> [(<parámetros>)] <instrucciones> end</pre>

Un aspecto importante a considerar es que la lista de parámetros, al igual que el uso de variables, **no** requiere definir el tipo de dato, pues este se determina en el momento de la llamada al método.

Es posible añadir los símbolos ! o ? al final del nombre de un método. ! indica que el método requiere más atención que la variante con el mismo nombre sin el !. Generalmente usado para indicar que se esta modificando al objeto sobre el cual es llamado el método. Son llamados también **métodos peligrosos**. Por ejemplo, es posible tener 2 métodos con el mismo nombre, pero el que termina en ! modifica al objeto y el otro (llamado **método seguro**) regresa una **copia** modificada del objeto. El símbolo ? indicaría que el método retorna un resultado booleano.

Otro aspecto interesante es que podemos regresar un valor en un método en Ruby con *return* o simplemente regresar el valor de la última línea de un método. Por lo que la instrucción *return* puede usarse únicamente para finalizaciones tempranas de un método.

En Ruby es posible que un método definido para una clase quede indefinido:

Sintaxis
<pre>undef <nombreMétodo> [(<parámetros>)]</pre>

Por ejemplo:

```
1 class Foo  
2     def foo  
3     end  
4 end  
5  
6 ...  
7  
8 undef foo
```

El uso de *undef* tiene que ser también dentro de la declaración de la clase, aunque sea en un momento posterior.

11.10.2 Un vistazo al acceso a métodos

Si bien en Ruby existen también los miembros públicos y privados, estos son usados para proveer acceso a los métodos, puesto que **los atributos son privados**. En Ruby, **los métodos son públicos por omisión** [?].

Métodos públicos

Accesibles donde la clase es visible.

Sintaxis
<code>public</code> <lista de métodos>

Métodos privados

Accesibles solo por instancias de la clase.

Sintaxis
<code>private</code> <lista de métodos>

La lista de métodos públicos o privados debe hacer referencia a métodos previamente definidos. Otra posibilidad es, similar a otros lenguajes, definirse el método en ese momento.

Ejemplo:

```
1 class A
2   private
3   def metodo_privado
4     # codigo
5   end
6 end
```

11.10.3 Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma. En Ruby los atributos son llamados variables de instancia, y no requieren ser declarados fuera de los métodos. Podemos usar variables de instancia conforme se necesiten. Una variable de instancia debe llevar como prefijo el símbolo @, por ejemplo:

@variableInstanciaOAtributo

Ejemplo:

```
1 class InstTest
2     def set_foo(n)
3         @foo = n
4     end
5     def set_bar(n)
6         @bar = n
7     end
8 end
```

El detalle es que en Ruby las variables de instancia no son accesibles externamente. Para poder accederlas debemos crear métodos de acceso como en el ejemplo pasado, o definir explícitamente el acceso a los atributos, lo que genera métodos con el nombre del atributo (pero sin el @). Existen 3 tipos de acceso a los atributos:

1. *attr_accessor*. Genera acceso de lectura y escritura a la variable de instancia.
2. *attr_reader*. Genera acceso de lectura sobre la variable de instancia.
3. *attr_writer*. Proporciona acceso de escritura sobre la variable de instancia.

Ejemplo:

```
1 class Usuario
2     attr_accessor :nombre
3     attr_accessor :apellidos
4     attr_reader :login
5     attr_writer :password
6     ...
7     def muestra
8         @nombre+" " +@apellidos
9     end
10 end
11
```

11.11 Objetos de clase en Ruby

En Ruby todos los números, arreglos, cadenas y demás entidades son objetos, lo que permite aplicar métodos definidos para esos objetos directamente a las literales:

```
1 10.succ           # 11
2 "hola".upcase     # "HOLA"
3 [2,1,5,3,4].sort  # [1,2,3,4,5]
4 objeto.metodo
```

Esto implicaría que cada objeto en Ruby es un objeto de alguna clase. De hecho es posible preguntarle a un objeto a que clase pertenece:

```
"hola".class
```

O mejor aún, los métodos que la clase tiene:

```
"hola".methods
```

Aunque esto puede hacerse directamente con el nombre de la clase⁸. Sin embargo el resultado puede variar, por ejemplo si le preguntamos a una clase a que clase pertenece! :

```
1 String.class
2
3 String.methods
```

11.11.1 Asignación de memoria al objeto

En Ruby, un objeto es instanciado mediante la ejecución del **método** *new*, el crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. El objeto instanciado típicamente es asignado a una variable.

Sintaxis
identificador = Clase.new

Un [ejemplo](#) completo:

⁸Para saber la razón por la que es posible usar el nombre de una clase, ver el tema de miembros de clase.

```
1 class Ejemplo3
2     @i  # no es necesario
3     @j  # especificar los atributos
4     def unMetodo x,y
5         @i=x
6         @j=y
7         puts "El valor de i es " + @i.to_s
8         puts "El valor de j es " + @j.to_s
9     end
10 end
11
12 require "Ejemplo3"
13 # ó load "Ejemplo3.rb"
14
15 obj1 = Ejemplo3.new
16 obj2 = Ejemplo3.new
17
18 obj1.unMetodo 10,20
19 obj2.unMetodo 100, 20
```

Listing 91. Ejemplo inicial de clases y objetos en Ruby.
Otro [ejemplo](#), una estructura de cola:

```
1 class Cola
2     @q=[]
3     @sloc= @rloc=-1
4
5     def ini
6         @sloc= @rloc=-1
7         @q=[]
8     end
9
10    def set val
11        if @sloc>1000
12            puts "La cola esta llena"
13            return false
14        end
15        @sloc+=1
16        @q[@sloc]=val
17        return true
18    end
```

```
19
20     def get
21         if @rloc == @sloc then
22             puts "La cola esta vacia"
23             return false
24         else
25             @rloc+=1
26             return @q[@rloc]
27         end
28     end
29 end
30 require "Cola"
31
32 a = Cola.new
33 b = Cola.new
34
35 a.ini
36 b.ini
37
38 a.set 1
39 b.set 2
40 a.set 11
41 b.set (22)
42
43 puts a.get
44 puts a.get
45 puts b.get
46 puts b.get()
47
48 a.get
```

Listing 92. Ejemplo 2 de clase en Ruby, una estructura simple de Cola.

11.12 Clases en Python

Una clase en **Python** consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase. Recordar que la indentación es usada en Python para definir bloques de código.

Sintaxis

```
class <nombre_clase> :  
    <cuerpo de la clase>
```

Ejemplo⁹:

```
1 class Ejemplo01:  
2     def fun(a,b):  
3         x=a  
4         y=b
```

11.13 Miembros de una clase en Python

11.13.1 Métodos miembro

Un método es una operación que pertenece a una clase. La sintaxis básica para declarar a un método:

Sintaxis

```
def nombreMétodo( <parámetros> ) :  
    <instrucciones>
```

⁹Algunos ejemplos como este no son programas completos, sino simples ejemplos de clases. Python es un lenguaje interpretado. No existe un método principal que inicie la ejecución. El intérprete recibe una lista de instrucciones y éste comienza ejecutando de la línea inicial hasta la última línea.

Un aspecto importante a considerar es que la lista de parámetros, al igual que el uso de variables, **no** requiere definir el tipo de dato, pues éste se determina en el momento de la llamada al método.

11.13.2 Atributos miembro

Los atributos o variables de instancia en Python deben ser declarados dentro de un método de la clase mediante:

```
self.nombre= valor
```

Los atributos podrán ser accedidos usando la misma notación (*self.nombre*)
Ejemplo:

```
1 class InstTest:
2     def set_foo(self, n):
3         self.foo = n
4     def set_bar(self, n):
5         self.bar = n
```

Es importante señalar que los métodos deben recibir al objeto en el primer parámetro. Esto se hace nombrando al primer parámetro como *self*.

11.13.3 Acceso a miembros en Python

Estrictamente hablando, no existen miembros realmente privados en Python, todos los miembros son considerados públicos. Sin embargo, en la práctica tenemos dos formas de declarar miembros privados.

Una opción entonces es declarar a los **elementos privados por convención**, declarando el nombre del miembro precedido por un guión bajo_. Entonces una variable *privada* deberá ser nombrada, por ejemplo, como *_foo*. Esto es una señal a los programadores de que se considera al miembro privado y no debería accederse fuera de la clase.

La otra opción es definir un *miembro privado* con un nombre precedido por doble guión bajo_. Entonces un miembro deberá de ser nombrado, por ejemplo *__foo*. Si el nombre de un atributo o método - de instancia o de clase- inicia con dos guiones bajos - y no termina con dos guiones bajos-, el miembro se considera privado. El resto de los miembros son considerados públicos [?].

Tener en cuenta que este es un mecanismo débil ya que aún se puede acceder a estos miembros *privados* a través de un truco conocido como *name mangling* a través del nombre de la clase (*obj.__nombreClase__atributo*). No se recomienda ya que va en contra del diseño orientado a objetos y puede llevar a un código más difícil de mantener y errores inesperados.

Ejemplo:

```

1 class Test:
2     def __init__(self):
3         self.__test=10
4         self.test=5
5     def getTest(self):
6         return self.__test
7
8 #main script
9 t= Test()
10 print(t.test)
11 print(t.getTest())
12 print(t._Test__test) # accediendo al atributo con el nombre de la clase
13
14 print(t.__test) #genera error
15

```

Listing 93. Ejemplo de miembros privados en Python.

Ahora bien, Python, de manera similar a Ruby, se adhiere al Principio de Acceso Uniforme establecido por Bertrand Meyer [?]: *"Todos los servicios ofrecidos por un módulo deben estar disponibles a través de una notación uniforme, que no traicione si se implementan mediante almacenamiento o mediante computación"*. En una forma más simple, establece que no debería haber ninguna diferencia sintáctica entre trabajar con un atributo, una propiedad calculada previamente o un método/consulta de un objeto.

Entonces de acuerdo a este principio, deberíamos acceder a los atributos directamente: $foo.x = 0$, en lugar de por medio de un método $foo.set_x(0)$. Acceder directamente al atributo nos permitiría hacer expresiones más cortas:

```

1 foo.x += 1

```

En lugar de:

```

1 foo.set_x(foo.get_x() + 1)

```

Sin embargo esto va en contra del principio del mínimo privilegio.

11.13.4 Acceso administrado a atributos en Python

Python tiene un protocolo de propiedades que nos permite dirigir acciones sobre atributos para operaciones tipo *get*, *set*, borrado y documentación básica.

Una propiedad es creada al asignarle el resultado de una función incorporada para un atributo de clase:

```
1 attribute = property(fget, fset, fdel, doc)
```

La siguiente clase usa una propiedad para trazar el acceso al atributo nombre. El dato es almacenado en `_nombre`.

```
1 class Persona:
2     def __init__(self, nombre):
3         self._nombre = nombre
4
5     def getNombre(self):
6         print('obteniendo...')
7         return self._nombre
8
9     def setNombre(self, val):
10        print('cambiando...')
11        self._nombre = val
12
13    def delNombre(self):
14        print('removiendo...')
15        del self._nombre
16    nombre = property(getNombre, setNombre, delNombre, "propiedad nombre ")
17
18    per1 = Persona('Bob Esponja')
19    print(per1.nombre)
20    per1.nombre = 'Roberto Esponja'
21    print(per1.nombre)
22    del per1.nombre      # Ejecuta delnombre
23
24    print('-'*20)
25    per2 = Persona('Susana F')
26    print(per2.nombre)
27    print(Persona.nombre.__doc__)  # O help(Persona.nombre)
28
```

Listing 94. Definiendo las propiedades de acceso de un tributo en Python.

Cuando el código es ejecutado, dos instancias reciben la propiedad del atributo, como sería con cualquier otro atributo definido en la clase.

Atributos calculados/validados

La ventaja de atributos administrados es que combinan la simplicidad de dar la impresión de acceder directamente al atributo al mismo tiempo que pueden hacer un cálculo o una validación.

```
1 class PropCuadrado:
2     def __init__(self, v):
3         self.val = v
4     def getX(self):          # Obtiene el atributo
5         return self.val ** 2
6     def setX(self, val):     # asigna en el atributo
7         self.val = val
8     X = property(getX, setX) # Sin propiedades del o docs
9
10 P= PropCuadrado(3)          # Dos instancias de la clase con property
11 Q= PropCuadrado(32)         # Cada una tiene un estado distinto
12
13 print(P.X)                  # 3 ** 2
14 P.X = 4
15 print(P.X)                  # 4 ** 2
16 print(Q.X)                  # 32 ** 2 (1024)
17
```

Listing 95. Uso clásico de métodos get y set en Python.

Definiendo propiedades de atributos con *Decorators*

Podemos también definir los métodos de acceso (*getter* y *setter*) y de borrado usando decoraciones (*decorators*)

Ejemplo:

```
1 class Foo:
2     def __init__(self):
3         self.__foo = 0
4
5     @property
6     def foo(self):
7         return self.__foo
8
9     @foo.setter
10    def foo(self, value):
```

```
11         self.__foo = value
12
13 f= Foo()
14 f.foo= "asignando a través del método setter"
15 print("Desplegando mediante el getter", f.foo)
```

Listing 96. Ejemplo de decorators en Python.

Veamos ahora el ejemplo inicial ahora con métodos decorados.

```
1 class Persona:
2     def __init__(self, nombre):
3         self._nombre = nombre
4
5     @property
6     def nombre(self):
7         "propiedad nombre"
8         print('obteniendo...')
9         return self._nombre
10
11     @nombre.setter
12     def nombre(self, val):
13         print('cambiando...')
14         self._nombre = val
15
16     @nombre.deleter
17     def nombre(self):
18         print('removiendo...')
19         del self._nombre
20
21 perl = Persona('Bob Esponja')
22 print(perl.nombre)
23 perl.nombre = 'Roberto Esponja'
24 print(perl.nombre)
25 del perl.nombre      # Runs delnombre
26
27 print('-'*20)      # sue inherits property too
28 per2 = Persona('Susana F')
29 print(per2.nombre)
30 print(Persona.nombre.__doc__)  # Or help(Persona.nombre)
31
```

Listing 97. Ejemplo de clase Persona con decorators en Python.

Definiendo propiedades de atributos con Descriptores

Otra opción para el manejo de atributos es el uso de descriptores (*descriptors*). En realidad es otra forma de describir las propiedades de los atributos pero ahora en una clase separada. Los descriptores son creados entonces como clases independientes, y asignados a atributos de clase. Como una propiedad, una clase descriptora maneja un único atributo, pero al tener su propio alcance (siendo clases) pueden manejar las propiedades para atributos más complejos de una mejor manera.

Sintaxis

```
class Descriptor:
    "inserta aqui cadena de descripción"
    def __get__(self, instance, owner): ... # regresa valor del atributo
    def __set__(self, instance, value): ... # regresa nada None
    def __delete__(self, instance): ... # regresa nada (None)
```

Ejemplo:

```
1 class Nombre:
2     "propiedad nombre"
3     def __get__(self, instancia, dueño):
4         print('obteniendo...')
5         return instancia._nombre
6     def __set__(self, instancia, val):
7         print('cambiando...')
8         instancia._nombre = val
9     def __delete__(self, instancia):
10        print('borrando...')
11        del instancia._nombre
12
13 class Persona:
14     def __init__(self, nombre):
15         self._nombre = nombre
16         nombre = Nombre()
17
```

```
18 perl = Persona('Bob Esponja')
19 print(perl.nombre)
20 perl.nombre = 'Roberto Esponja'
21 print(perl.nombre)
22 del perl.nombre
23
24 print('-'*20)
25 per2 = Persona('Susana F')
26 print(per2.nombre)
27 print(Nombre.__doc__)
28
29
```

Listing 98. Ejemplo de clase Persona con descriptores en Python.
Inclusive se puede poner como clase anidada:

```
1 class Persona:
2     def __init__(self, nombre):
3         self._nombre = nombre
4     class Nombre:
5         "propiedad nombre"
6         def __get__(self, instancia, dueño):
7             print('obteniendo...')
8             return instancia._nombre
9         def __set__(self, instancia, val):
10            print('cambiando...')
11            instancia._nombre = val
12        def __delete__(self, instancia):
13            print('borrando...')
14            del instancia._nombre
15
16    nombre = Nombre()
17
18 #script de prueba
19 perl = Persona('Bob Esponja')
20 print(perl.nombre)
21 perl.nombre = 'Roberto Esponja'
22 print(perl.nombre)
23 del perl.nombre
24 print('-'*20)
25 per2 = Persona('Susana F')
26 print(per2.nombre)
```

```
27 print(Persona.Nombre.__doc__)
```

```
28
```

Listing 99. Ejemplo de clase Persona con descriptores en clase anidada en Python.

Manejo genérico de atributos

También es posible manejar el acceso a los atributos redefiniendo los métodos especiales de acceso `getattr`, `setattr` y `delattr`.

```
1 class Persona:
2     def __init__(self, nombre):
3         self._nombre = nombre           #lanza __setattr__
4
5     def __getattr__(self, attr):
6         print('get: ' + attr)
7         if attr == 'nombre':           #intercepta nombre
8             return self._nombre        # no se cicla al ser el atributo real
9         else:                           #otros son errores
10            raise AttributeError(attr)
11
12    def __setattr__(self, attr, val):
13        print('set: ' + attr)
14        if attr == 'nombre':
15            attr = '_nombre'             #asigna nombre interno
16            self.__dict__[attr] = val    #evita ciclado recursivo de __setattr__
17
18    def __delattr__(self, attr):
19        print('del: ' + attr)
20        if attr == 'nombre':
21            attr = '_nombre'             #evita ciclado recursivo
22            del self.__dict__[attr]      #aunque mucho menos común
23
24    per1 = Persona('Bob Esponja')
25    print(per1.nombre)                   #Corre __getattr__
26    per1.nombre = 'Roberto Esponja'     #Corre __setattr__
27    print(per1.nombre)
28    del per1.nombre                      #Corre __delattr__
29
30    print('-'*20)
31    per2 = Persona('Susana F')
32    print(per2.nombre)
```

33

Listing 100. Ejemplo de clase Persona con manejo genérico de atributos en Python.

11.14 Objetos de clase en Python

En Python todos los números, arreglos, cadenas y demás entidades son objetos. Cada objeto tiene un valor, un identificador y un tipo. El identificador del objeto y el tipo pueden ser accedidos mediante las operaciones *id()* y *type()* respectivamente:

```

1 >>> n=10
2 >>> id(n)
3 3492368
4 >>> type(n)
5 <class 'int'>
6 >>> arr=[2,3,4]
7 >>> id(arr)
8 19201080
9 >>> type(arr)
10 <class 'list'>

```

11.14.1 Asignación de memoria al objeto

En Python, un objeto es instanciado mediante la ejecución objeto de la clase que se quiere instanciar, éste crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. El objeto instanciado típicamente es asignado a una variable.

Sintaxis

```
identificador = Clase()
```

Un [ejemplo](#) completo:

```

1 class Ejemplo3:
2     def unMetodo(self, x, y):
3         self.i=x
4         self.j=y
5         print("El valor de i es " + str(self.i))

```



```
6         print("El valor de j es " + str(self.j))
7     # script de ejecucion
8     obj1= Ejemplo3()
9     obj2= Ejemplo3()
10    obj1.unMetodo(10, 20)
11    obj2.unMetodo(100, 200)
```

Listing 101. Ejemplo de clases en Python.

Otro [ejemplo](#), una estructura de cola:

```
1 class Cola:
2     def ini(self):
3         self.sloc= self.rloc=-1
4         self.q=[]
5
6     def setC(self, val):
7         if self.sloc>1000:
8             print("La cola esta llena")
9             return False
10        self.sloc+=1
11        self.q.append(val)
12        return True
13
14    def getC(self):
15        if self.rloc == self.sloc:
16            print("La cola esta vacia")
17            return False
18        else:
19            self.rloc+=1
20            return self.q[self.rloc]
21
22    # script de ejecucion
23    a = Cola()
24    b = Cola()
25
26    a.ini()
27    b.ini()
28
29    a.setC(1)
30    b.setC(2)
31    a.setC(11)
```

```
32 b.setC(22)
33
34 print(a.getC())
35 print(a.getC())
36 print(b.getC())
37 print(b.getC())
38
39 a.getC()
```

Listing 102. Ejemplo.

11.15 Clases en Scala

Una clase en **Scala** consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase.

Sintaxis

```
class <nombre_clase> {  
    <cuerpo de la clase>  
}
```

Ejemplo:

```
1 class Ejemplo01 {  
2     def fun(a:int,b:int) {  
3         var x: int =a  
4         val y: int =b  
5         ...  
6     }  
7 }
```

11.16 Miembros de una clase en Scala

11.16.1 Métodos miembro

Un método es una operación que pertenece a una clase. La sintaxis básica para declarar a un método:

Sintaxis

```
def nombreMétodo [( <parámetros> ) ][: tipo retorno] =  
    <instrucciones>
```

La lista de parámetros es opcional. Un método sin parámetros no requiere los paréntesis en su uso (definición y/o llamada):

```
def foo = 'foo'
```

Atributos miembro

Los atributos Scala deben ser declarados dentro la clase mediante:

Sintaxis
<code>var nombre: tipo [= valor]</code>

Los atributos podrán ser accedidos usando la misma notación.

Es posible también definir atributos que no van a cambiar (su referencia, aunque el objeto referenciado pueda cambiar -similar a elementos final en Java-):

Sintaxis
<code>val nombre: tipo [= valor]</code>

Ejemplo:

```
1 class InstTest {  
2   var foo: Int  
3   var bar: Int  
4 }
```

Scala puede inferir el tipo de dato cuando estos son inicializados:

```
var s= "cadena"
```

11.16.2 Un vistazo al acceso a miembros

La declaración de visibilidad o acceso a miembros es similar a la de Java y C#. Sin embargo, los miembros son considerados **públicos por omisión** y no existe un modificador para este tipo de acceso.

Miembros públicos

Visibles donde la clase sea visible. No se tiene una palabra reservada para especificar al miembro como público.

Miembros privados

Solamente son accesible dentro de la clase (o tipo) que lo define, y sus tipos anidados. Accesibles únicamente dentro del mismo paquete.

Sintaxis
<code>private</code> <definición de miembro>

Existen otras opciones de accesibilidad que serán comentadas más adelante. Ejemplo:

```
1 class Fecha {  
2     private var dia: Int  
3     private var mes, an: Int  
4  
5     def setDia(d: Int): Boolean = {}  
6     def getDia(): Int = {}  
7     def setMes(m: Int): Boolean = {}  
8     def getMes(): Int = {}  
9     def setAn(a: Int): Boolean = {}  
10    def getAn(): Int = {}  
11 }
```

11.17 Objetos de clase en Scala

Scala es un lenguaje puro orientado a objetos. Todos los números, arreglos, cadenas y demás entidades son objetos. A diferencia de Java, inclusive los números y las funciones son considerados objetos.

11.17.1 Números como objetos

```
1 scala> 1  
2 res1: Int = 1  
3  
4 scala> 1  
5 res2: Int = 1  
6  
7 scala> 1+2  
8 res3: Int = 3  
9
```

```

10 scala> (1).+(2)
11 res4: Int = 3

```

11.17.2 Funciones como objetos

Puesto que Scala considera a las funciones como objetos, es posible pasar las funciones como argumentos en los parámetros, almacenarlos en variables y regresarlos como resultado de otras funciones. El hecho de que las funciones sean consideradas *first-class citizens* es lo que hace que Scala puede ser considerado también un lenguaje con capacidades de **programación funcional**!

11.17.3 Asignación de memoria al objeto

En Scala, un objeto es instanciado igual que en Java, usando la instrucción *new* seguida del nombre de la clase. El objeto instanciado típicamente es asignado a una variable. La variable puede o no haber sido definida con anterioridad. La excepción sería en los objetos que corresponden a valores primitivos en Java, para los cuales no es necesario usar específicamente el constructor. De hecho, tratar de usar un constructor para estos tipos generará un error de compilación.

Sintaxis

```
identificador = new Clase()
```

Un [ejemplo](#) completo:

```

1  class Ejemplo3 {
2      private var i: Int = 0
3      private var j: Int = 0
4      def unMetodo(x: Int, y: Int) = {
5          i = x
6          j = y
7          println("El valor de i es " + i)
8          println("El valor de j es " + j)
9      }
10 }
11
12 object HolaMundo {
13     def main(args: Array[String]) = {
14         var obj1: Ejemplo3 = new Ejemplo3
15         var obj2 = new Ejemplo3
16     }

```

```
17     obj1.unMetodo(10, 20)
18     obj2.unMetodo(100, 200)
19 }
20 }
```

Listing 103. Ejemplo de clase en Scala.

Otro [ejemplo](#), una estructura de cola:

```
1  class Cola {
2      private var sloc, rloc = 0
3      private var q = new Array[Int] (0) //0 elementos
4
5      def ini() = {
6          sloc = -1
7          rloc = -1
8          q= new Array[Int] (10) //10 elementos
9      }
10
11     def setC(valor: Int):Boolean = {
12         if (sloc>1000){
13             println("La cola esta llena")
14             return false
15         }
16         sloc+=1
17         q(sloc)=valor
18         return true
19     }
20
21     def get :Int ={
22         if (rloc == sloc){
23             println("La cola esta vacia")
24             return -1
25         } else {
26             rloc += 1
27             return q(rloc)
28         }
29     }
30 }
31
32 object ColaTest {
33     def main(args : Array[String]) = {
```

```
34     val a,b: Cola = new Cola
35
36     a.ini()
37     b.ini()
38
39     a.setC(1)
40     b.setC(2)
41     a.setC(11)
42     b.setC(22)
43
44     println(a.get)
45     println(a.get)
46     println(b.get)
47     println(b.get)
48
49     a.get
50 }
51 }
```

Listing 104. Ejemplo de clases en Scala, una estructura de Cola simple.

11.18 Clases en C#

La definición en C#, de manera similar a C++ y Java, consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves.

Sintaxis

```
class <nombre_clase> {  
    <cuerpo de la clase>  
}
```

Ejemplo:

```
1 public class Ejemplo01      {  
2     int x;  
3     float y;  
4     void fun(int a, float b) {  
5         x=a;  
6         y=b;  
7     }  
8 }
```

11.19 Miembros de una clase en C#

Los miembros en C# son esencialmente los atributos y los métodos de la clase.

11.19.1 Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

La sintaxis mínima es la siguiente:

Sintaxis

```
tipo nombreAtributo;
```

Los atributos pueden ser inicializados desde su lugar de declaración: `tipo nombreAtributo = valor;`
o, en el caso de variables de objetos: `tipo nombreAtributo = new Clase();`

11.19.2 Métodos miembro

Un método es una operación que pertenece a una clase. **No** es posible declarar métodos fuera de la clase. Además, no existe el concepto de método prototipo como en C++.

Sin embargo, igual que en C++, la declaración de una función ó método miembro es considerada dentro del ámbito de su clase.

La sintaxis básica para declarar a un método:

Sintaxis

```
tipoRetorno nombreMétodo ( [<parámetros>] ) {  
    <instrucciones>  
}
```

11.19.3 Tipos de parámetros

Existen 4 tipos de paso de parámetros en C#. Pasó de parámetros por valor, por referencia, parámetros de salida y arreglos de parámetros.

- **Pasó de parámetros por valor.** Este es el tipo de parámetros por omisión. Se manda una copia del valor, sea un tipo de dato simple (o una enumeración o estructura), o una copia de la referencia al objeto.
- **Por referencia.** Parámetros por referencia usan la palabra clave *ref*, y no usan los valores de las variables, sino las variables mismas. Ejemplo:

```
1 void foo(ref MiClase c) {  
2     c=null;  
3 }
```

La modificación de *c* permanece después de la llamada al método *foo*.

- **Parámetros de salida.** Parámetros de salida usan la palabra clave *out*, y no usan los valores de las variables, sino las variables mismas. Es similar al parámetro por referencia pero se considera como un parámetro sin asignación en la llamada al método, y el método debe asignarle un valor antes de terminar su ejecución.

- **Arreglos de parámetros.** Permiten pasar un número variable de parámetros en una llamada a un método. Utiliza el modificador *params* en la definición del método. El número de parámetros puede ser de cero a n, respetando el tipo de dato del parámetro. Ejemplo:

```

1 void muestra (params int[] num) {
2     foreach (int x in num)
3     {
4         Console.Write (x+" ");
5     }
6     Console.WriteLine();
7 }

```

Llamando al método:

```

1 muestra(1, 2, 3, 4, 5);
2 int[] arr = {99, 88, 77};
3 muestra(arr);

```

11.19.4 Un vistazo al acceso a miembros

En C#, como en Java, se define el acceso a cada miembro de manera unitaria, al contrario de la definición de acceso por grupos de miembros de C++. Miembros en clases son considerados privados por omisión¹⁰.

Miembros públicos

Sintaxis
public <definición de miembro>

Miembros privados

Sintaxis
private <definición de miembro>

¹⁰Accessibility Levels (C# Reference) ([C# Reference](#))

Miembros internos

Miembros definidos de esta manera son accesibles por métodos de cualquier clase que se encuentre en el mismo *assembly*¹¹.

Sintaxis

```
internal <definición de miembro>
```

Recordatorio

Es una buena costumbre de programación acceder a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo. Estos métodos de acceso y modificación comúnmente tienen el prefijo *get* y *set*, respectivamente.

Ejemplo:

```
1 public class Fecha{  
2     private int dia;  
3     private int mes, an;  
4  
5     public bool setDia(int d){}    //poner día  
6     public int getDia()           {} //devuelve día  
7     public bool setMes(int m){}  
8     public int getMes(){}  
9     public bool setAn(int a) {}  
10    public int getAn() {}  
11 }
```

11.20 Objetos de clase en C#

Como en Java, en C# todos los objetos son creados dinámicamente, por lo que se necesita reservar la memoria de estos en el momento en que se van a ocupar. El operador es también *new*.

¹¹Un *assembly* es una colección de archivos compilados reunidos como un archivo ejecutable o un DLL.

11.20.1 Asignación de memoria al objeto

El operador *new* crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. Esta referencia debe de recibirla un identificador de la misma clase de la que se haya reservado la memoria.

Sintaxis

`<identificador> = new Clase();`
o en el momento de declarar a la variable de objeto: `Clase <identificador> = new Clase();`

El concepto de *new* va asociado de la noción de constructor, pero esta se verá más adelante, por el momento basta con adoptar esta sintaxis para poder completar ejemplos de instanciación.

Un [ejemplo](#) completo:

```
1 using System;
2
3 namespace Application
4 {
5     public class Ejemplo3 {
6         public int i, j;
7
8         static void Main() {
9             Ejemplo3 e3= new Ejemplo3();
10            Ejemplo3 e1= new Ejemplo3();
11
12            e1.i=10;
13            e1.j=20;
14            e3.i=100;
15            e3.j=20;
16
17            Console.WriteLine(e1.i);
18            Console.WriteLine(e3.i);
19        }
20    }
21 }
```

Listing 105. Ejemplo de clase en C#.

Otro [ejemplo](#), una estructura de cola:

```
1 using System;
2
3 namespace Application
4 {
5
6     class Cola{
7         private int [] q;
8         private int sloc, rloc;
9
10        public void ini() {
11            sloc=rloc=-1;
12            q=new int[10];
13        }
14
15        public bool set(int val){
16            if(sloc>=10){
17                Console.WriteLine("la cola esta llena");
18                return false;
19            }
20            sloc++;
21            q[sloc]=val;
22            return true;
23        }
24
25        public int get(){
26            if(rloc==sloc) {
27                Console.WriteLine("la cola esta vacia");
28                return -1;
29            }
30            else {
31                rloc++;
32                return q[rloc];
33            }
34        }
35    }
36
37    public class PruebaCola {
38        static void Main(){
39
40            Cola a= new Cola(); // new crea realmente el objeto
41            Cola b= new Cola(); // reservando la memoria
```

```
42         Cola pCola= new Cola();
43
44         //Inicializacion de los objetos
45         a.ini();
46         b.ini();
47         pCola.ini();
48
49         a.set(1);
50         b.set(2);
51         pCola.set(3);
52         a.set(11);
53         b.set(22);
54         pCola.set(33);
55
56         Console.WriteLine(a.get());
57         Console.WriteLine(a.get());
58         Console.WriteLine(b.get());
59         Console.WriteLine(b.get());
60         Console.WriteLine(pCola.get());
61         Console.WriteLine(pCola.get());
62     }
63 }
64 }
```

Listing 106. Ejemplo 2 de clase en C#, una estructura simple de Cola.

11.21 Clases en D

La definición de una en **D** consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre `{}`. Recordar que **D** es un lenguaje sintácticamente similar a **C++**.

Sintaxis

```
class <nombre_clase> {  
    <cuerpo de la clase>  
}
```

Ejemplo:

```
1 class Ejemplo1 {  
2     int x;  
3     float y;  
4     void fun(int a, float b) {  
5         x=a;  
6         y=b;  
7     }  
8 }
```

11.22 Miembros de una clase en D

Al igual que en **C++**, en **D** una clase está formada por un conjunto de miembros que pueden ser datos, funciones, clases anidadas, enumeraciones, tipos de dato, etc.

Ejemplo:

```
1 class Ejemplo2{  
2     int i;  
3     int i; //error  
4     int j;  
5     void func() {}  
6 }
```


11.22.1 Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

11.22.2 Métodos miembro

Los métodos al igual que los atributos, deben ser definidos en la clase. A diferencia de C++, la definición completa del método debe ir dentro de la clase.

Ejemplo:

```
1 //código en ejemplo3.h
2 class Ejemplo3 {
3     public:
4     int x;
5     float y;
6     int funX(int a) {
7         x=a;
8         return x;
9     }
10 }
```

11.22.3 Un vistazo al acceso a miembros

Miembros públicos

Se utiliza cuando queremos dar a usuarios de una clase (no implementadores) el acceso a miembros de esa clase, los miembros deben ser declarados públicos.

Sintaxis

```
public:
    <definición de miembros>
```

Miembros privados

Si queremos ocultar ciertos miembros de una clase de los usuarios de la misma, debemos declarar a los miembros como privados. Por omisión los miembros se consideran privados.

Sintaxis

private:

<definición de miembros>

Ejemplo:

```

1 class Fecha {
2     private:
3     int dia;
4     int mes;
5     int an;
6
7     public:
8     char setDia(int d) { ... } //poner día
9     int getDia() { ... } //devuelve día
10    char setMes(int m) { ... }
11    int getMes() { ... }
12    char setAn(int a) { ... }
13    int getAn() { ... }
14 }
```

11.23 Objetos de clase en D

Vamos a ver como es posible crear objetos o instancias de clase en D. En D, un objeto es una variable de un tipo definido por el usuario[?]. Para indicar al recolector de basura que un objeto se puede liberar, es posible ejecutar *objeto.destroy()*¹².

Un **ejemplo** completo:

```

1 module Ejemplo3;
2 import std.stdio;
3
4 class Ejemplo3 {
```

¹²Anteriormente se usaba la instrucción *delete*, pero ha sido discontinuada en las versiones actuales del compilador: *Deprecation: The delete keyword has been deprecated. Use object.destroy() (and core.memory.GC.free() if applicable) instead.* 1

```
5         public:
6             int i;
7             int j;
8     }
9
10    void main(string[] args) {
11        Ejemplo3 e1= new Ejemplo3;
12        Ejemplo3 e2= new Ejemplo3();
13        e1.i=10;
14        e1.j=20;
15        e2.i=100;
16        e2.j=20;
17        writeln(e1.i);
18        writeln(e2.i);
19    }
```

Listing 107. Ejemplo de clase en D.

Otro [ejemplo](#), una cola:

```
1 module Cola;
2
3 import std.stdio;
4
5 class Cola{
6     private:
7         int [10] q;
8         int sloc, rloc;
9     public:
10        void ini() {
11            sloc=rloc=-1;
12        }
13
14        char set(int val){
15            if(sloc>=10){
16                writeln("la cola esta llena");
17                return 0;
18            }
19            sloc++;
20            q[sloc]=val;
21            return 1;
22        }
23    }
```

```
23
24     int get() {
25         if(rloc==sloc)
26             writeln("la cola esta vacia");
27         else {
28             rloc++;
29             return q[rloc];
30         }
31         return 0;
32     }
33 }
34
35 //cola definida en un arreglo
36 int main() {
37     Cola a,b, pCola= new Cola;
38     a= new Cola;
39     b= new Cola;
40
41     a.ini();
42     b.ini();
43     pCola.ini();
44     a.set(1);
45     b.set(2);
46     pCola.set(3);
47     a.set(11);
48     b.set(22);
49     pCola.set(33);
50     writeln(a.get());
51     writeln(a.get());
52     writeln(b.get());
53     writeln(b.get());
54     writeln(pCola.get());
55     writeln(pCola.get());
56     pCola.destroy();
57     return 0;
58 }
```

Listing 108. Ejemplo 2 de clase en D, una estructura de Cola simple.

11.24 Usando la palabra reservada *this* en C++, C#, D, Scala y Java

Cuando en algún punto dentro del código de algunos de los métodos se quiere hacer referencia al objeto ligado en ese momento con la ejecución del método, podemos hacerlo usando la palabra reservada *this*.

Una razón para usarlo es querer tener acceso a algún atributo posiblemente oculto por un parámetro del mismo nombre.

También puede ser usado para regresar el objeto a través del método, sin necesidad de realizar una copia en un objeto temporal.

La sintaxis es la misma en C++ y en Java, con la única diferencia del manejo del operador de indirección "*" si, por ejemplo, se quiere regresar una copia y no la referencia del objeto. **D**, **C#** y **Scala** también utilizan *this*.

Ejemplo en C++:

```
1 Fecha Fecha::getFecha() {
2     return *this;
3 }
```

Ejemplo en Java:

```
1 class Fecha {
2     private int dia;
3     private int mes, an;
4     ...
5     public Fecha getFecha() {
6         return this;
7     }
8     ...
9 }
```

11.25 Usando la palabra reservada *self* en Ruby

En Ruby, para hacer referencia en un método al objeto donde es ejecutado el método (el mismo que recibe el mensaje) se usa la palabra reservada *self*, en lugar de *this*.

Ejemplo:

```
1  class SelfEjemplo
2      @x=0
3      @y=0
4      def getSelfEjemplo
5          return self
6      end
7      attr_accessor :x, :y
8  end
9
10 #código de prueba
11 obj1= SelfEjemplo.new
12
13 obj1.x=10
14 puts obj1.x
15 obj1.y="YY"
16 puts obj1.y
17
18 obj2=obj1.getSelfEjemplo
19
20 puts obj2.x
21
22 obj1.y=123
23 puts obj1.y # despliega 123
24 puts obj2.y # despliega 123
```

Listing 109. Ejemplo de uso de *self* en Ruby.

11.26 Usando *self* en Python

En Python, el primer argumento de un método es llamado *self*. Esto no es más que una convención puesto que no se trata de una palabra reservada. Es utilizado para identificar al objeto que ejecuta al método y es necesario inclusive para el uso de sus atributos y métodos, como se ha visto en ejemplos anteriores.

Ejemplo:

```
1
2 class SelfEjemplo:
3     def ini(self):
4         self.x=0
5         self.y=0
6
7     def getSelfEjemplo(self):
8         return self
9
10
11 # script de ejecucion
12 obj1 = SelfEjemplo()
13 obj1.ini()
14
15 obj1.x=10
16 print(obj1.x)
17 obj1.y="yy"
18 print(obj1.y)
19
20 obj2=obj1.getSelfEjemplo()
21
22 print(obj2.x)
23
24 obj1.y=123
25 print(obj1.y) #despliega 123
26 print(obj2.y) #despliega 123
27
```

Listing 110. Ejemplo de uso de *self* en Python.

Capítulo 12

Polimorfismo AdHoc: Sobrecarga de operaciones

12.1 Introducción

Es posible tener el **mismo nombre** para una operación con la condición de que tenga parámetros diferentes. La diferencia debe de ser al menos en el tipo de datos. Al menos un parámetro debe ser diferente.

Concepto
Si se tienen dos o más operaciones con el mismo nombre y diferentes parámetros se dice que dichas operaciones están sobrecargadas .

El compilador sabe que operación ejecutar a través de la **firma** de la operación, que es una combinación del nombre de la operación y el número y tipo de los parámetros.

El tipo de regreso de la operación puede ser igual o diferente.

La sobrecarga¹ de operaciones sirve para hacer un código más legible y modular. La idea es utilizar el mismo nombre para operaciones relacionadas. Si no tienen nada que ver entonces es mejor utilizar un nombre distinto. A continuación ejemplos de sobrecarga en C++ y Java.

¹También conocida como homonimia.

12.2 Ejemplos de sobrecarga en C++

```
1 class MiClase{
2     int x;
3
4     public:
5     void modifica() {
6         x++;
7     }
8     void modifica(int y){
9         x=y*y;
10    }
11 }
```

Ejemplo completo en C++:

```
1 //fuera de P00
2 #include <iostream>
3 using namespace std;
4
5 int cuadrado(int i){
6     return i*i;
7 }
8 double cuadrado(double d){
9     return d*d;
10 }
11
12 int main() {
13     cout<<"10 elevado al cuadrado: "<<cuadrado(10)<<endl;
14     cout<<"10.5 elevado al cuadrado: "<<cuadrado(10.5)<<endl;
15     return 0;
16 }
```

Listing 111. Ejemplo de sobrecarga en C++.

12.3 Ejemplo de sobrecarga en Java

```
1 class MiClase{
2     int x;
```

```
3      public          void modifica() {
4          x++;
5      }
6      public void modifica(int y){
7          x=y*y;
8      }
9  }
```

12.4 En Ruby no hay sobrecarga

Ruby **no** soporta sobrecarga de operaciones. En Ruby la firma de un método es su nombre. Si dos métodos son definidos con el mismo nombre, la última implementación definida es la que se espera.

```
1  class MiClase
2      attr_reader :x
3
4      def modifica
5          @x+=1
6      end
7      def modifica y
8          @x=y*y
9      end
10
11 end
12
13 #prueba
14 mc=MiClase.new
15
16 puts mc.x
17 # mc.modifica    -- Error pues se ha sobreescrito el método
18
19 mc.modifica 10
20 puts mc.x
```

Listing 112. Ejemplo en ruby, redefiniendo un método.

12.5 Sobrecarga de operaciones en Python

Originalmente Python no contaba con sobrecarga de operaciones. De hecho tratar de sobrecargar de forma tradicional nos llevaría a una situación como la del siguiente ejemplo:

```
1 class MiClase:
2     def modifica(self):
3         self.x+=1
4
5     def modifica(self, y):
6         self.x=y*y
7
8     #prueba
9 mc=MiClase()
10 mc.modifica(5)
11 print(mc.x)
12 # mc.modifica()    -- Error pues se ha sobreescrito el metodo
13 mc.modifica(10)
14 print(mc.x)
```

Listing 113. Ejemplo redefiniendo un método en Python.

Sin embargo, una aproximación al concepto de sobrecarga ha sido implementado en versiones más recientes a través decoraciones declaradas dentro del módulo *functools*

12.5.1 Sobrecarga de funciones con *singledispatch*

A partir de la versión 3.4 de Python, el decorador *singledispatch* permitirá convertir una función tradicional en una función sobrecargada. Es importante señalar que únicamente es posible hacerlo con al tipo del primer argumento de la función. Ejemplo:

```
1
2 from functools import singledispatch
3
4 @singledispatch
5 def suma(a, b):
6     raise NotImplementedError('Tipo no soportado')
7
8 @suma.register(int)
```

```

9  def _(a, b):
10     print("El primer argumento es de tipo: ", type(a))
11     print(a + b)
12
13  @suma.register(str)
14  def s(a, b):          # nombre de la función no es relevante
15     print("El primer argumento es de tipo: ", type(a))
16     print(a + b)
17
18  @suma.register
19  def _(a: list, b):    #el tipo de dato puede ir especificado como "anotación"
20     print("El primer argumento es de tipo: ", type(a))
21     print(a + b)
22
23  suma(1, 2)
24  suma('Programación', 'Python')
25  suma([1, 2, 3], [5, 6, 7])
26
27  print('Tipos de datos manejados: ', suma.registry.keys())
28

```

Listing 114. Ejemplo sobrecargando una función con *singledispatch*.

De manera similar a otros lenguajes, la implementación del método a enviarse a ejecución es el del tipo que corresponda, en este caso al tipo del primer argumento. Es importante señalar que el nombre de la función que se registra con no es relevante.

En el siguiente código se ve que el decorador permite apilamiento para que una función soporte más de un tipo.

```

1
2  from functools import singledispatch
3  from decimal import Decimal
4
5  @singledispatch
6  def suma(a, b):
7      raise NotImplementedError('Tipo no soportado')
8
9  @suma.register(float)
10 @suma.register(Decimal)
11 def _(a, b):
12     print("El primer argumento es de tipo: ", type(a))
13     print(a + b)

```

```
14
15 suma(1.23, 5.5)
16 suma(Decimal(100.5), Decimal(10.789))
17
18 print('Tipos de datos manejados: ', suma.registry.keys())
19
```

Listing 115. Ejemplo sobrecargando una función con *singledispatch* y apilamiento de decoradores.

12.5.2 Sobrecarga de métodos con *singledispatchmethod*

La solución pasada es útil para funciones pero no para métodos de instancia o de clase, debido a que el primer argumento aquí es para el objeto o la clase - *self* o *cls* -. A partir de Python 3.8, se puede utilizar *singledispatchmethod* para sobrecargar métodos, métodos de clase, métodos abstractos o estáticos para el primer argumento del método (después de *self* o *cls*).

```
1
2 from functools import singledispatchmethod
3
4 class Negacion:
5     @singledispatchmethod
6     def neg(self, arg):
7         raise NotImplementedError("No puedo negar este tipo de dato")
8
9     @neg.register
10    def _(self, arg: int):
11        return -arg
12
13    @neg.register
14    def _(self, arg: bool):
15        return not arg
16
17
18 obj = Negacion()
19
20 print(obj.neg(10))
21 print(obj.neg(True))
22
```

Listing 116. Ejemplo sobrecargando un método con *singledispatchmethod*.

```
1
2 from functools import singledispatchmethod
3
4 class Negacion:
5     @singledispatchmethod
6     @classmethod
7     def neg(cls, arg):
8         raise NotImplementedError("No puedo negar este tipo de dato")
9
10    @neg.register
11    @classmethod
12    def _(cls, arg: int):
13        return -arg
14
15    @neg.register
16    @classmethod
17    def _(cls, arg: bool):
18        return not arg
19
20
21 obj = Negacion()
22
23 print(obj.neg(10))
24 print(Negacion.neg(True))
25
```

Listing 117. Ejemplo sobrecargando un método de clase con *singledispatchmethod*.

12.6 Ejemplo de sobrecarga en Scala

```
1 class MiClase {
2     var x: Int=0
3     def modifica() {
4         x+=1
5     }
6
7     def modifica(y:Int) {
8         x=y*y
9     }
10 }
```

```
9      }
10   }
11
12   object Sobrecarga {
13       def main(args: Array[String]) = {
14           System.out.println("Sobrecarga")
15           var mc /* : MiClase */ = new MiClase()
16           mc.modifica(5)
17           println(mc.x)
18           mc.modifica()
19           println(mc.x)
20           mc.modifica(10)
21           println(mc.x)
22       }
23   }
```

Listing 118. Ejemplo de sobrecarga en Scala.

12.7 Ejemplo de sobrecarga en C#

```
1   using System;
2
3   namespace Application
4   {
5
6       class MiClase {
7           public int x=0;
8           public void modifica() {
9               x+=1;
10           }
11
12           public void modifica(int y) {
13               x=y*y;
14           }
15       }
16
17       //public class Sobrecarga {
18       class Sobrecarga {
19           static void Main() {
20               Console.WriteLine("Sobrecarga");
```



```
21         MiClase mc = new MiClase();
22         mc.modifica(5);
23         Console.WriteLine(mc.x);
24         mc.modifica();
25         Console.WriteLine(mc.x);
26         mc.modifica(10);
27         Console.WriteLine(mc.x);
28     }
29 }
30
31 }
```

Listing 119. Ejemplo de sobrecarga en C#.

12.8 Ejemplo de sobrecarga en D

```
1 module Sobrecarga;
2
3 import std.stdio;
4
5
6 class MiClase {
7     public:
8         int x=0;
9         void modifica() {
10             x+=1;
11         }
12
13         void modifica(int y) {
14             x=y*y;
15         }
16 }
17
18 //public class Sobrecarga {
19 int main() {
20     writeln("Sobrecarga");
21     MiClase mc = new MiClase();
22     mc.modifica(5);
23     writeln(mc.x);
24     mc.modifica();
```

```
25     writeln(mc.x);  
26     mc.modifica(10);  
27     writeln(mc.x);  
28     return 0;  
29 }
```

Listing 120. Ejemplo de sobrecarga en D.

Capítulo 13

Constructores y destructores

13.1 Constructores y destructores en C++

Con el manejo de los tipos de datos primitivos, el compilador se encarga de reservar la memoria y de liberarla cuando estos datos salen de su ámbito.

En la programación orientada a objetos, se trata de proporcionar mecanismos similares, aunque con mayor funcionalidad. Cuando un objeto es creado es llamado un método conocido como **constructor**, y al salir se llama a otro conocido como **destructor**. Si no se proporcionan estos métodos se asume la acción más simple.

13.1.1 Constructor

Un **constructor** es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato y si puede permitir la homonimia o sobrecarga.

Ejemplo:

```
1  class Cola{
2      private:
3          int q[100];
4          int sloc, rloc;
5
6      public:
7          Cola( );           //constructor
8          void put(int);
9          int get( );
10 };
11
12 //implementación del constructor
13 Cola::Cola ( ) {
14     sloc=rloc=0;
15     cout<<"Cola inicializada \n";
16 }
```

Un constructor si puede ser llamado desde un método de la clase.

Lista de inicialización de atributos Constructor

En C++, se pueden inicializar los atributos de una clase después de los parámetros del constructor utilizando una lista de inicialización. La lista de inicialización se coloca después de la lista de parámetros del constructor y antes del cuerpo del constructor.

Aquí se presenta un ejemplo:

```
1
2  #include <iostream>
3
4  class MiClase {
5  public:
6      // Constructor con parámetros
7      MiClase(int parametro1, int parametro2)
8          : atributo1(parametro1), atributo2(parametro2) {
9          // Cuerpo del constructor (si es necesario)
10         // Puedes realizar más inicializaciones o lógica aquí
11     }
12
13     // Otros métodos y miembros de la clase
14
15 private:
16     int atributo1;
17     int atributo2;
18 };
19
20 int main() {
21     // Crear un objeto de la clase MiClase e inicializar sus atributos
22     MiClase objeto(10, 20);
23
24     return 0;
25 }
26
```

Listing 121. Ejemplo de lista de inicialización de atributos en un constructor en C++.

El constructor de *MiClase* toma dos parámetros enteros y utiliza una lista de inicialización para asignar esos valores a los atributos *atributo1* y *atributo2*. Esto es una práctica recomendada en C++ porque puede ayudar a mejorar el rendimiento y evitar ambigüedades en la inicialización de los miembros de la clase, pero tiene como desventaja que no incluye la validación de los mismos.

Constructor de Copia

Es útil agregar a todas las clases un constructor de copia que reciba como parámetro un objeto de la clase y copie sus datos al nuevo objeto.

C++ proporciona un constructor de copia por omisión, sin embargo es una **copia a nivel de miembro** y puede no realizar una copia exacta de lo que queremos. Por ejemplo en casos de punteros a memoria dinámica, se tendría una copia de la dirección y no de la información referenciada.

Sintaxis

```
<nombre clase>(const <nombre clase> &<objeto>);
```

Ejemplo:

```
1  //ejemplo de constructor de copia
2  #include <iostream>
3  #include <time.h>
4  #include <stdlib.h>
5
6  using namespace std;
7
8  class Arr{
9      private:
10         int a[10];
11     public:
12
13         Arr(int x=0) {
14             for( int i=0; i<10; i++){
15                 if (x==0)
16                     x=rand();
17                 a[i]=x;
18             }
19         }
20
21         Arr(const Arr &copia){    //constructor de copia
22             for( int i=0; i<10; i++)
23                 a[i]=copia.a[i];
24         }
25
26         char set(int, int);
27         int get(int) const ;
28         int get(int);
29     };
30
31     char Arr::set(int pos, int val ){
32         if(pos>=0 && pos<10){
33             a[pos]=val;
34             return 1;
35         }
36         return 0;
37     }
```

```
38
39 int Arr::get(int pos) const {
40     if(pos>=0 && pos<10)
41         return a[pos];
42     // a[9]=0; error en un metodo constante
43     return 0;
44 }
45
46 int Arr::get(int pos) { //no es necesario sobrecargar
47     if(pos>=0 && pos<10) // si el metodo no modifica
48         return a[pos];
49     return 0;
50 }
51
52 int main() {
53     Arr a(5), b;
54     srand( time(nullptr) );
55
56     a.set(0,1);
57     a.set(1,11);
58     cout<<a.get(0)<<endl;
59     cout<<a.get(1)<<endl;
60
61     b.set(0,2);
62     b.set(1,22);
63     cout<<b.get(0)<<endl;
64     cout<<b.get(1)<<endl;
65
66     Arr d(a);
67     cout<<d.get(0)<<endl;
68     cout<<d.get(1)<<endl;
69
70     return 0;
71 }
```

Listing 122. Ejemplo de constructor de copia en C++.

13.1.2 Destructor

La contraparte del constructor es el **destructor**. Este se ejecuta momentos antes de que el objeto sea destruido, ya sea porque salen de su ámbito o por medio de una instrucción *delete*. El uso más común para un destructor es liberar la memoria

asignada dinámicamente, aunque puede ser utilizado para otras operaciones de finalización, como cerrar archivos, una conexión a red, etc.

El destructor tiene al igual que el constructor el nombre de la clase pero con una tilde como prefijo (~).

El destructor tampoco regresa valores ni tiene parámetros.

Ejemplo:

```
1 class Cola{
2     private:
3     int q[100];
4     int sloc, rloc;
5
6     public:
7     Cola( );           //constructor
8     ~Cola();          //destructor
9     void put(int);
10    int get( );
11 };
12
13 Cola::~~Cola( ){
14     cout<<"cola destruida\n";
15 }
```

Ejemplo completo de Cola con constructor y destructor:

```
1 //cola definida en un arreglo
2 //incluye constructores y destructores de ejemplo
3 #include <iostream>
4 #include <string.h>
5 #include <stdio.h>
6
7 using namespace std;
8
9 class Cola{
10     private:
11     int q[10], sloc, rloc;
12     char *nom;
13
14     public:
15     Cola(const char *cad=NULLPTR) { //funcion en linea
16         if(cad){ //cadena!=NULLPTR
```



```
17         nom=new char[strlen(cad)+1];
18         strcpy(nom, cad);
19     }else
20         nom=nullptr;
21     sloc=rloc=-1;
22 }
23 ~Cola( ) {
24     if(nom){ //nom!=nullptr
25         cout<<"Cola : "<<nom<<" destruida\n";
26         delete [] nom;
27     }
28 }
29
30 char set(int);
31 int get();
32 };
33
34 char Cola::set(int val){
35     if(sloc>=10){
36         cout<<"la cola esta llena";
37         return 0;
38     }
39     sloc++;
40     q[sloc]=val;
41     return 1;
42 }
43 int Cola::get(){
44     if(rloc==sloc)
45         cout<<"la cola esta vacia";
46     else {
47         rloc++;
48         return q[rloc];
49     }
50     return 0;
51 }
52
53 int main(){
54     Cola a("Cola a"),b("Cola b"),
55         *pCola= new Cola("Cola dinamica pCola");
56     a.set(1);
57     b.set(2);
58     pCola->set(3);
```

```
59     a.set (11);
60     b.set (22);
61     pCola->set (33);
62     cout<<a.get () <<endl;
63     cout<<a.get () <<endl;
64     cout<<b.get () <<endl;
65     cout<<b.get () <<endl;
66     cout<<pCola->get () <<endl;
67     cout<<pCola->get () <<endl;
68
69     delete pCola;
70 }
```

Listing 123. Ejemplo completo de Cola con constructor y destructor en C++.

13.1.3 Punteros Inteligentes (Smart Pointers)

En C++ moderno (a partir de C++11), se recomienda el uso de **punteros inteligentes** en lugar de la gestión manual de memoria con *new* y *delete*. Los punteros inteligentes implementan el principio RAII (*Resource Acquisition Is Initialization*), liberando automáticamente la memoria cuando el objeto sale de su ámbito, evitando fugas de memoria y simplificando el código.

Los punteros inteligentes están definidos en la biblioteca `<memory>` y son:

- **std::unique_ptr** - Posesión exclusiva de un recurso. No puede ser copiado, solo movido.
- **std::shared_ptr** - Posesión compartida con conteo de referencias. Múltiples punteros pueden apuntar al mismo objeto.
- **std::weak_ptr** - Referencia no propietaria que no incrementa el contador de referencias. Usado para romper ciclos de referencias.

Ejemplo con `std::unique_ptr`:

```
1 // Ejemplo de uso de std::unique_ptr
2 #include <iostream>
3 #include <memory>
4 #include <string.h>
5
6 using namespace std;
7
```

```
8  class Cola{
9      private:
10         int q[10], sloc, rloc;
11         char *nom;
12
13     public:
14         Cola(const char *cad=nullptr) {
15             if(cad){
16                 nom=new char[strlen(cad)+1];
17                 strcpy(nom, cad);
18             }else
19                 nom=nullptr;
20             sloc=rloc=-1;
21         }
22         ~Cola( ) {
23             if(nom){
24                 cout<<"Cola : "<<nom<<" destruida\n";
25                 delete [] nom;
26             }
27         }
28
29         char set(int);
30         int get();
31     };
32
33     char Cola::set(int val){
34         if(sloc>=10){
35             cout<<"la cola esta llena";
36             return 0;
37         }
38         sloc++;
39         q[sloc]=val;
40         return 1;
41     }
42
43     int Cola::get(){
44         if(rloc==sloc)
45             cout<<"la cola esta vacia";
46         else {
47             rloc++;
48             return q[rloc];
49         }
```

```
50         return 0;
51     }
52
53     int main() {
54         Cola a("Cola a"), b("Cola b");
55
56         // Uso de unique_ptr - gestion automatica de memoria
57         unique_ptr<Cola> pCola = make_unique<Cola>("Cola pCola");
58
59         a.set(1);
60         b.set(2);
61         pCola->set(3); // acceso con ->
62         a.set(11);
63         b.set(22);
64         pCola->set(33);
65
66         cout<<a.get()<<endl;
67         cout<<a.get()<<endl;
68         cout<<b.get()<<endl;
69         cout<<b.get()<<endl;
70         cout<<pCola->get()<<endl;
71         cout<<pCola->get()<<endl;
72
73         // No es necesario delete - el destructor se llama automaticamente
74         // cuando pCola sale de ambito
75     }
```

Listing 124. Ejemplo de uso de `std::unique_ptr` para gestión automática de memoria en C++.

Ejemplo con `std::shared_ptr`:

```
1 // Ejemplo de std::shared_ptr - posesion compartida
2 #include <iostream>
3 #include <memory>
4
5 using namespace std;
6
7 class Recurso {
8     public:
9     Recurso(int v) : valor(v) {
10         cout << "Recurso creado: " << valor << endl;
11     }
```

```
12     ~Recurso() {
13         cout << "Recurso destruido: " << valor << endl;
14     }
15     void mostrar() {
16         cout << "Valor: " << valor << endl;
17     }
18     private:
19     int valor;
20 };
21
22 int main() {
23     // Crear shared_ptr
24     shared_ptr<Recurso> ptr1 = make_shared<Recurso>(42);
25
26     cout << "Contador de referencias: " << ptr1.use_count() << endl;
27
28     {
29         // Compartir posesion
30         shared_ptr<Recurso> ptr2 = ptr1;
31         cout << "Contador de referencias: " << ptr1.use_count() << endl;
32         ptr2->mostrar();
33
34         // ptr2 sale de ambito aqui
35     }
36
37     cout << "Contador de referencias: " << ptr1.use_count() << endl;
38
39     // El recurso se destruye automaticamente cuando el último
40     // shared_ptr sale de ambito
41     return 0;
42 }
```

Listing 125. Ejemplo de `std::shared_ptr` mostrando el conteo de referencias en C++.

Ventajas de los Punteros Inteligentes

- Gestión automática de memoria - no es necesario llamar a *delete*.
- Prevención de fugas de memoria (*memory leaks*).
- Prevención de doble liberación (*double delete*).
- Código más seguro y expresivo.
- Compatible con excepciones - la memoria se libera incluso si ocurre una excepción.

13.2 Constructores y finalizadores en Java

En Java, cuando un objeto es creado es llamada un método conocido como **constructor**, y al salir se llama a otro conocido como **finalizador**¹. Si no se proporcionan estos métodos se asume la acción más simple.

13.2.1 Constructor

Un constructor es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato de retorno y si puede permitir la homonimia o sobrecarga, y la modificación de acceso al mismo.

Ejemplo:

```
1 public class Cola{
2     private int q[];
3     private int sloc, rloc;
4     public void put(int){ ... }
5     public int get( ){ ... }
6     //      implementación del constructor
7     public Cola ( ) {
8         sloc=rloc=0;
9         q= new int[100];
10        System.out.println("Cola inicializada ");
11    }
12 }
```

El constructor se ejecuta en el momento de asignarle la memoria a un objeto, y es la razón de usar los paréntesis junto al nombre de la clase al usar la instrucción *new*:

```
Fecha f = new Fecha(10,4,2007);
```

Si no se especifica un constructor, Java incluye uno predeterminado, que asigna memoria para el objeto e inicializa las variables de instancia a valores predeterminados. Este constructor se omite si se especifica uno o más por parte del programador.

¹En C++ no existe el concepto de finalizador, sino el de destructor, porque su tarea primordial es liberar la memoria ocupada por el objeto, cosa que no es necesario realizar en Java.

13.2.2 Finalizador

La contraparte del constructor en Java es el método *finalize* o finalizador. Este se ejecuta momentos antes de que el objeto sea destruido por el recolector de basura. El uso más común para un finalizador es liberar los recursos utilizados por el objeto, como una conexión de red o cerrar algún archivo abierto.

No es muy común utilizar un método finalizador, más que para asegurar situaciones como las mencionadas antes. El método iría en términos generales como se muestra a continuación²:

Sintaxis
<pre>protected void finalize() { <instrucciones> }</pre>

El finalizador puede ser llamado como un método normal, inclusive puede ser sobrecargado, pero un finalizador con parámetros no puede ser ejecutado automáticamente por la máquina virtual de Java. Se recomienda evitar el definir un finalizador con parámetros.

²No se ha mencionado el modificador *protected*. Este concepto se explicará una vez que se haya visto el manejo de herencia.

13.3 Inicializadores en Ruby

En Ruby, el método que podemos usar para inicializar un objeto es llamado *initialize*. No es llamado constructor porque, en Ruby, son internamente dos procesos separados. El método *initialize* no es un constructor y es definido automáticamente como privado.

Concepto

La explicación para esto es que las clases en Ruby son instancias de la clase *Class*. Por cada clase definida, un objeto de tipo *Class* es creado y asignado a una constante del nombre especificado en la declaración de la clase. Cuando el método *new* es llamado (*NombreClase.new*) para crear un objeto, se ejecuta por default el método *new* de *Class*, el cual ejecuta al método *allocate* para asignar la memoria del objeto, y por último, el método *initialize* del nuevo objeto es ejecutado.

Sintaxis

```
class NombreClase
  def initialize [(lista de parámetros)]
    <código>
  end
  ...
end
...
obj = NombreClase.new [(parámetros)]
```

Ejemplo:

```
1 class Cola
2     def initialize
3         @sloc= @rloc=-1
4         @q=[]
5     end
6     ...
7 end
```

13.4 Inicializando y eliminando en Python

Clases en Python no tienen constructores ni destructores explícitos. Lo más parecido a un constructor es el método `__init__`. El concepto es similar al de inicialización de objetos en Ruby. El método `__init__` está implementado por omisión y no estamos obligados a definirlo. Podemos añadir atributos en otros métodos sin necesidad de inicializarlos en el método *init*:

```

1 class Rectángulo:
2     def area(self) -> float:
3         return self.largo * self.ancho
4
5 r = Rectángulo()
6 r.largo, r.ancho = 11, 9
7 print(r.area())

```

Listing 126. Ejemplo mostrando el uso de atributos sin añadirlos en *init*.

Sin embargo, no se considera una buena práctica pues puede generar confusión y errores. Lo mejor es inicializar los atributos en el método *init*.

Sintaxis

```

class NombreClase:
    def __init__(self, [<parametros>]):
        <código>
        ...

obj = NombreClase([<parámetros>])

```

Ejemplo:

```

1 class Cola:
2
3     def __init__(self)
4         self.sloc= self.rloc=-1
5         self.q=[]
6     end

```

```
7         ...
8     end
```

Por otro lado, el método `__del__()` es lo más parecido que se tiene a un destructor, más parecido realmente al finalizador de java, ya que es llamado cuando todas las referencias a un objeto se han eliminado y éste es recogido por el recolector de basura.

```
1  #Ejemplo de inicializador y eliminador en Python
2  class Empleado:
3      # Inicializador
4      def __init__(self):
5          print('Objeto empleado creado.')
6
7      # Eliminando (Llamando al 'destructor')
8      def __del__(self):
9          print('Destructor llamado, Empleado borrado.')
10
11  obj = Empleado()
12  del obj
```

Listing 127. Ejemplo mostrando el uso de los métodos `__init__()` y `__del__()` en Python.

13.5 Constructores en Scala

Scala distingue entre un constructor primario y constructores secundarios. El constructor primario en realidad se forma con la definición de la clase seguida de una lista de parámetros. Una clase en Scala con su sección de parámetros es llamada clase **polimórfica**, de otro modo, una clase sin parámetros es una clase **monomórfica**. Estos parámetros de clase pueden ser usados directamente dentro de la definición de la misma y ya no es necesario definirlos como atributos.

Sintaxis

```
class NombreClase (<parametros>) {
  ...
  obj = new NombreClase([<parámetros>])
```

Esto permite ahorrar espacio y tener una clase potencialmente más concisa. Pero ¿Qué pasa si yo necesito validar los valores de los parámetros? Dentro del código de la clase es posible insertar dicho código y formará parte del constructor primario.

Ejemplo³:

```
1 class Cola(var sloc: Int, var rloc: Int, var q: Array[Int]) {
2     if (q==null) {
3         sloc=-1
4         rloc=-1
5     }
6
7 }
```

Es posible tener constructores auxiliares de la forma:
TABLA con más de una línea de CÓDIGO

³Si no se especifica un identificador de tipo var, se toma como de tipo val por omisión

Sintaxis

```
def this([<parámetros>])  
  <instrucciones>
```

13.6 Constructores y destructores en C#

Constructores y destructores pueden ser definidos en C# de forma similar a C++.

13.6.1 Constructor

Un constructor en C# es definido especificando un método del mismo nombre de la clase.

Sintaxis

```
class NombreClase {  
    NombreClase([<parametros>]) {  
        <código>  
    }  
}  
...  
obj = new NombreClase([<parámetros>]);
```

Ejemplo:

```
1 class Cola{  
2     private int q[];  
3     private int sloc, rloc;  
4     public void put(int){  
5         //...  
6     }  
7     public int get(){  
8         //...  
9     }  
10    //      implementación del constructor  
11    public Cola () {  
12        sloc=rloc=0;  
13        q= new int[100];  
14        Console.WriteLine("Cola inicializada ");  
15    }  
16 }
```

13.6.2 Destructor

C# cuenta con un recolector de basura, por lo que el uso del destructor no es tan frecuente como en C++. Aún así es posible definir un destructor usando la misma sintaxis de C++:

```
1 ~Cola( ){  
2     Console.WriteLine("cola destruida");  
3 }
```

Su comportamiento en realidad es más parecido al finalizador de Java.

13.7 Constructores y destructores en D

D incluye también el concepto de y destructor de objetos.

13.7.1 Constructor

Aunque los miembros de una clase son inicializados en D por su valor por omisión o inicializándolos estáticamente en el momento de la definición del miembro, es posible definir un constructor en D es definido especificando un método con el nombre de *this*.

Sintaxis

```
class NombreClase {
    this([<parametros>]) {
        <código>
    }
}
...
obj = new NombreClase([<parámetros>]);
```

Ejemplo:

```
1 class Cola{
2     private:
3     int q[];
4         int sloc, rloc;
5     public:
6     void put(int) {
7         //...
8         }
9         int get ( ) {
10        //...
11        }
12        // implementación del constructor
13        this ( ) {
14            sloc=rloc=0;
15            q= new int[100];
```

```
16         writeln("Cola inicializada ");
17     }
18 }
```

13.7.2 Destructor

D cuenta con un recolector de basura, sin embargo al igual que C# cuenta con la opción de definir un destructor.

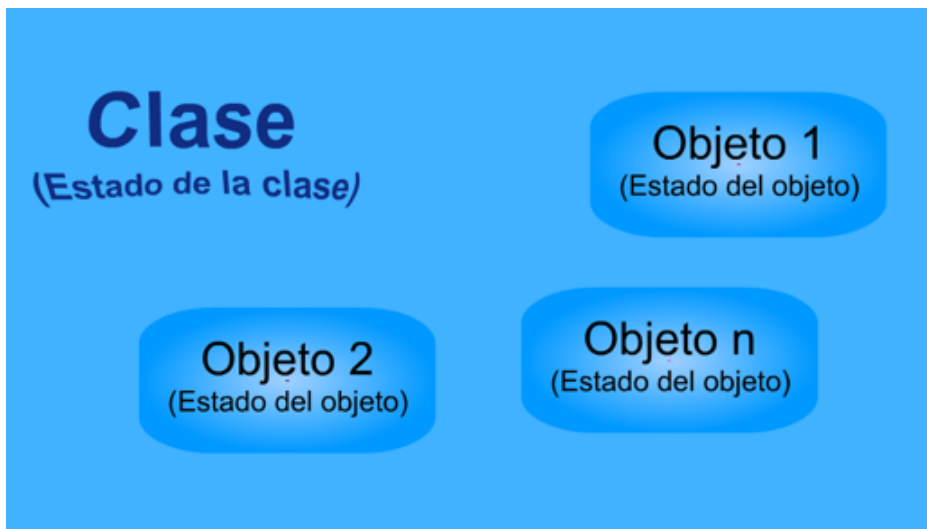
```
1 ~this( ){
2     writeln("cola destruida");
3 }
```

Capítulo 14

Miembros de clase o estáticos

Cada objeto tiene su propio estado, pero a veces es necesario tener valores por clase y no por objeto. En esos casos se requiere tener atributos **estáticos** que sean compartidos por todos los objetos de la clase.

Concepto
Existe solo una copia de un miembro estático y no forma parte de los objetos de la clase. Este tipo de miembro son también conocidos como miembros de clase.



14.1 Miembros estáticos en C++

Un miembro estático es accesible desde cualquier objeto de la clase o mediante el operador de resolución de alcance binario (::) y el nombre de la clase, dado que el miembro estático **existe** aunque no haya instancias de la clase.

Sin embargo, el acceso sigue restringido bajo las reglas de acceso a miembros:

- Si se quiere acceder a un miembro estático que es privado deberá hacerse mediante un método público.
- Si no existe ninguna instancia de la clase entonces deberá ser por medio de un método público y estático.

Además, un método estático solo puede tener acceso a miembros estáticos.

Ejemplo:

```
1  class Objeto{
2      private:
3          char nombre[10];
4          static int numObjetos;
5      public:
6          Objeto(char *cadena=nullptr);
7          ~Objeto();
8  };
9
10 Objeto::Objeto(const char *cadena){
11     if(cadena!=nullptr)
12         strcpy(nombre, cadena);
13     else
14         nombre=nullptr;
15     numObjetos++;
16 }
17
18 Objeto::~~Objeto(){
19     numObjetos--;
20 }
```

Los atributos estáticos deben de ser inicializados al igual que los atributos constantes, fuera de la declaración de la clase. Por ejemplo:

```
int Clase::atributo=0;
```

```
int const Clase::ATRCONST=50;
```

Ejemplo:

```
1  //prueba de miembros estáticos
2  #include <iostream>
3  #include <stdio.h>
4  #include <string.h>
5
6  using namespace std;
7
8  class Persona{
9      private:
10         static int nPersonas;
11         static const int MAX;
12         char *nombre;
13
14     public:
15     Persona(const char *c=nullptr) {
16         if(c!=nullptr) {
17             nombre= new char[strlen(c)+1];
18             strcpy(nombre, c);
19             cout<<"Persona: "<<nombre<<endl;
20         }else{
21             nombre=nullptr;
22             cout<<"Persona: "<<endl;
23         }
24         nPersonas++;
25     }
26
27     ~Persona() {
28         cout<<"eliminando persona : "<<nombre<<endl;
29         if(nombre)
30             delete []nombre;
31         nPersonas--;
32     }
33
34     static int getMax() {
35         return MAX;
36     }
37
38     static int getnPersonas() {
39         return nPersonas;
40     }
```

```
41 };
42 int Persona::nPersonas=0;
43 const int Persona::MAX=10;
44
45 int main() {
46
47     cout<<"Máximo de personas: "<<Persona::getMax()<<endl;
48     cout<<"Número de personas: "<<Persona::getnPersonas()<<endl;
49
50     Persona per1;
51     cout<<"Máximo de personas: "<<Persona::getMax()<<endl;
52     cout<<"Número de personas: " <<Persona::getnPersonas()<<endl;
53
54     Persona per2("persona 2");
55     cout<<"Máximo de personas: "<<per2.getMax()<<endl;
56     cout<<"Número de personas: "<<per2.getnPersonas()<<endl;
57     return 0;
58 }
```

Listing 128. Ejemplo miembros estáticos en C++.

14.2 Miembros estáticos en Java

Un miembro estático en Java se maneja de la misma forma que en C++. Cada uno de los objetos tiene su propio estado independiente del resto de los objetos, compartiendo al mismo tiempo un estado común al tener todos los objetos acceso al estado de la clase, el cual es único y existe de forma independiente.

Ejemplo:

```
1 public class Objeto{
2     private String nombre;
3     private static int numObjetos;
4     public Objeto(String cadena){
5         if(cadena.length() !=0)
6             nombre=cadena;
7         else
8             nombre="cadena por omision";
9         numObjetos++;
10    }
11
12    public static int getNumObjetos(){
13        return numObjetos;
14    }
15
16    public static void main(String argv[]) {
17        System.out.println("Objetos: " + getNumObjetos());
18        System.out.println("Objetos: " + Objeto.getNumObjetos());
19        Objeto uno,dos;
20        uno= new Objeto("");
21        dos= new Objeto("Objeto dos");
22        System.out.println("Objetos: " + uno.getNumObjetos());
23        System.out.println("Objetos: " + dos.getNumObjetos());
24    }
25 }
```

Listing 129. Ejemplo de miembros estáticos en Java.

Una diferencia que se puede apreciar en este ejemplo con respecto a C++, es que no estamos obligados a inicializar el elemento estático, pues éste es inicializado automáticamente.

14.3 Miembros de clase en Ruby

Los elementos estáticos en Ruby son conocidos como miembros de clase, el cual es un término también usado en otros lenguajes debido a que el ámbito de los miembros va más allá del objeto pues existen en la clase sin depender de un objeto en particular y representan, como ya mencionamos, un estado de la clase.

14.3.1 Atributos de clase

Las variables de clase en Ruby son nombradas usando @@ como prefijo y deben ser inicializadas antes de ser usadas en definiciones de métodos. Referenciar un atributo de clase no inicializado genera un error. Las variables de clase son compartidas entre descendientes de las clases donde estas fueron definidas.

14.3.2 Métodos de clase

Los métodos de clase son definidos en Ruby usando como prefijo el nombre de la clase o la palabra reservada *self* seguido de punto y el nombre del método de clase. *self* se resuelve como el nombre de la clase en este caso, siendo un comportamiento distinto a cuando es usado dentro de un método (*self* se resuelve como el objeto en ejecución).

Ejemplo:

```
1 class Persona
2     @@nPersonas=0
3     @@MAX=100      # no es una constante
4     @nombre
5     def initialize nom
6         @nombre=nom
7         puts "Persona: " + @nombre if nom.size>0
8         @@nPersonas+=1
9     end
10
11     def self.getMax
12         return @@MAX
13     end
14
15     def Persona.getnPersonas
16         return @@nPersonas
17     end
18
19 end
```



```
20
21 #código principal
22
23 puts "Numero maximo de personas: #{Persona.getMax}"
24 puts "Numero de personas: #{Persona.getnPersonas}"
25
26 perl = Persona.new "Persona 1"
27 puts "Numero maximo de personas: #{Persona.getMax}"
28 puts "Numero de personas: #{Persona.getnPersonas}"
29
30 per2 = Persona.new "Persona 2"
31 puts Persona.getMax
32 puts Persona.getnPersonas
33 # puts per2.getMax    -- error
34
```

Listing 130. Ejemplo de miembros de clase en Ruby.

A diferencia de C++ y Java, en Ruby un método de clase no puede ser accedido mediante un objeto. Un método de clase solo puede ser usado externamente mediante el nombre de la clase.

14.4 Miembros estáticos / de clase en Python

En Python existe el concepto de atributos de clase y métodos de clase, pero estos últimos se diferencian de los métodos estáticos como se explica a continuación.

14.4.1 Atributos de clase

En Python los **atributos de clase** en realidad son los atributos que se definen **dentro de la clase**. Los atributos de instancia (*data attributes* en Python) son definidos dentro del método `__init__`.

14.4.2 Método de clase y Método estático

En Python existe una ligera diferencia entre un método de clase y un método estático. Son definidos usando *decorators*: `@staticmethod` y `@classmethod`. Un método de clase recibe a la clase como primer argumento, de la misma forma que un método de instancia recibe a la instancia como primer argumento, por lo que tiene el ámbito de la clase. Como en Java, el método puede ser llamado mediante el nombre de la clase o el de un objeto de la clase. El método estático por su parte, no recibe un argumento implícito¹, por lo que **no puede modificar ni el estado de un objeto ni el de la clase**. Un [ejemplo](#) marcando la diferencia:

```
1 class A:
2     def metodo(self, x):
3         print ("ejecutando metodo(%s,%s)"%(self,x))
4
5     @classmethod
6     def metodo_de_clase(cls, x):
7         print ("ejecutando metodo de clase(%s,%s)"%(cls,x))
8
9     @staticmethod
10    def metodo_estatico(x):
11        print ("ejecutando metodo estatico(%s)"%x)
12
13 #Script de ejecución
14 a=A()
15
16 #La llamada normal a un método de instancia.
17 #El objeto es pasado de manea implícita como primer argumento
18 a.metodo(1)
```

¹[Class method vs static method 2015](#)

```

19 # salida: ejecutando metodo(<__main__.A object at 0xb7dbef0c>,1)
20
21 # En un metodo de clase, la clase del objeto es pasada
22 # implicitamente como primer argumento
23 a.metodo_de_clase(1)
24 # salida: ejecutando metodo de clase(<class '__main__.A'>,1)
25
26 # Tambien puede ejecutarse un metodo de clase usando
27 # el nombre de la clase
28 A.metodo_de_clase(1)
29 # salida: ejecutando metodo de clase(<class '__main__.A'>,1)
30
31 # Un metodo estatico no pasa implicitamente ni al objeto
32 # ni la clase como primer argumento
33 a.metodo_estatico(1)
34 # salida: ejecutando metodo estatico(1)
35
36 #metodo esta confinado al objeto a
37 print(a.metodo)
38
39 #metodo_de_clase esta confinado a la clase A, no a al objeto a
40 print(a.metodo_de_clase)
41
42 # metodo_estatico no esta confinado ni al objeto
43 # ni a la clase, no va dicha informacion como argumento
44 print(a.metodo_estatico)
45 print(A.metodo_estatico)

```

Listing 131. Ejemplo de métodos estáticos y de clase en Python.

Ejemplo:

```

1 class Persona:
2     nPersonas=0
3     MAX=100      # no es una constante
4     def __init__(self, nom):
5         self.nombre=nom
6         print("Persona: " + self.nombre)
7         Persona.nPersonas+=1
8
9     @classmethod
10    def getMax(cls):

```

```
11         return cls.MAX
12
13     @classmethod
14     def getnPersonas(cls):
15         return cls.nPersonas
16
17     def getnPersonas2(self):
18         return self.nPersonas
19
20     #código principal
21
22     print("Numero maximo de personas:", Persona.getMax() )
23     print("Numero de personas:", Persona.getnPersonas() )
24
25     per1 = Persona("Persona 1")
26     print("Numero maximo de personas: ", Persona.getMax() )
27     print("Numero de personas: ", Persona.getnPersonas() )
28
29     per2 = Persona("Persona 2")
30     print(Persona.getMax() )
31     print(per2.getnPersonas() )
32     print(per2.getnPersonas2() )
```

Listing 132. Ejemplo de métodos de clase en Python.

Ejemplo:

```
1 from datetime import date
2
3 class Persona:
4     def __init__(self, nombre, edad):
5         self.nombre = nombre
6         self.edad = edad
7
8     # método de clase crea una clase persona por año de nacimiento.
9     @classmethod
10    def porAñoNacimiento(cls, nombre, año):
11        return cls(nombre, date.today().year - año)
12
13    # método estático para verificar si una persona es un adulto
14    @staticmethod
15    def esAdulto(edad):
```

```
16         return edad > 18
17
18     persona1 = Persona('un nombre', 30)
19     persona2 = Persona.porAñoNacimiento('otro nombre', 1999)
20
21     print("Edad: ", persona1.edad)
22     print("Edad: ", persona2.edad)
23
24     print("¿Es adulto?", Persona.esAdulto(22))
25
```

Listing 133. Ejemplo de métodos de clase y estático en Python.

14.5 Miembros estáticos en Scala

Scala no cuenta con miembros estáticos. Una opción es utilizar la definición de miembros “estáticos” en objetos únicos (*singleton*). Un ejemplo de este tipo de estructura se vio en el ejemplo de inicial del lenguaje:

```
1 object HolaMundo {  
2   def main(args: Array[String]) = {  
3     println("Hola, Mundo!")  
4   }  
5 }
```

Listing 134. Ejemplo de objeto único *singleton* en Scala.

Donde *HolaMundo* es un objeto único que por lo tanto no tiene que ser instanciado.

14.6 Miembros estáticos / de clase en C#

Un miembro estático en C# se maneja de forma similar a C++ y Java. Cada uno de los objetos tiene su propio estado independiente del resto de los objetos. También, los miembros estáticos sirven para definir un estado de clase. Sin embargo, los objetos no tienen acceso dicho estado, por lo que los miembros estáticos únicamente pueden ser accedidos usando el nombre de la clase.

Ejemplo:

```
1 using System;
2 class Objeto{
3     private String nombre;
4     private static int numObjetos;
5     public Objeto(String cadena){
6         if(cadena.Length!=0)
7             nombre=cadena;
8         else
9             nombre="cadena por omision";
10        numObjetos++;
11    }
12
13    public static int getNumObjetos(){
14        return numObjetos;
15    }
16
17    public static void Main() {
18        Console.WriteLine("Objetos: " + getNumObjetos());
19        Console.WriteLine("Objetos: " + Objeto.getNumObjetos());
20
21        Objeto uno,dos;
22
23        uno= new Objeto("");
24        dos= new Objeto("Objeto dos");
25        // valido en Java y C++ pero no en C#
26        //Console.WriteLine("Objetos: " + uno.getNumObjetos());
27        //Console.WriteLine("Objetos: " + dos.getNumObjetos());
28        // error CS0176: Static member 'Objeto.getNumObjetos()' cannot be accessed
29        Console.WriteLine("Objetos: " + getNumObjetos());
30    }
31 }
```

Listing 135. Ejemplo de miembros estáticos en C#.

Otro ejemplo:

```
1  //prueba de miembros estáticos
2  using System;
3
4  class Persona{
5      private static int nPersonas;
6      // private static const int MAX=10;
7      // constantes son estaticas por default
8      private const int MAX=10;
9
10     string nombre;
11
12     public Persona(string c){
13         if(c.Length!=0)
14             // no se proporciona un constructor que reciba un string
15             //nombre= new string(c);
16             nombre= (string) c.Clone(); // o c.Copy() Ver diferencia entre copia
17
18         else
19             nombre="";
20         Console.WriteLine("Persona: "+nombre);
21         nPersonas++;
22     }
23
24     public static int getMax(){
25         return MAX;
26     }
27     public static int getnPersonas(){
28         return nPersonas;
29     }
30
31     public static void Main() {
32         Console.WriteLine("Maximo de personas: "+Persona.getMax());
33         Console.WriteLine("Numero de personas: "+Persona.getnPersonas());
34
35         Persona per1= new Persona("");
36         Console.WriteLine("Maximo de personas: "+Persona.getMax());
37         Console.WriteLine("Numero de personas: "+Persona.getnPersonas());
38
39         Persona per2= new Persona("persona 2");
40         //Console.WriteLine("Maximo de personas: "+per2.getMax());
```



```
41         //Console.WriteLine("Numero de personas: "+per2.getnPersonas());
42         Console.WriteLine("Maximo de personas: "+Persona.getMax());
43         Console.WriteLine("Numero de personas: "+Persona.getnPersonas());
44
45
46         Persona per3= new Persona("persona 3");
47         Console.WriteLine("Maximo de personas: "+Persona.getMax());
48         Console.WriteLine("Numero de personas: "+Persona.getnPersonas());
49     }
50 }
```

Listing 136. Ejemplo 2 de miembros estáticos en C#.

Al igual que en Java, no estamos obligados a inicializar el atributo estático.

El manejo de cadenas aquí cambió ligeramente debido a que C# no proporciona un constructor del tipo *string(string)*. Se usó el método *Clone()* el cual genera una copia superficial del objeto. En este caso una copia profunda *Copy()* hubiera dado el mismo resultado. Otro aspecto importante en cuanto a cadenas es que en C# las cadenas son inmutables, por lo que si se requiere un cambio a una cadena debe generarse en una nueva. La restricción es para hacer un manejo más eficiente de las cadenas.

14.7 Miembros estáticos / de clase en D

El manejo de miembros estáticos es similar al de C++

Ejemplo:

```
1 module Objeto;
2 import std.stdio;
3
4 class Objeto{
5     private:
6         string nombre;
7         static int numObjetos;
8     public:
9         this(string cadena){
10             if(cadena.length!=0)
11                 nombre=cadena;
12             else
13                 nombre="cadena por omision";
14             numObjetos++;
15         }
16
17         static int getNumObjetos(){
18             return numObjetos;
19         }
20     }
21 int main() {
22     writeln("Objetos: ", Objeto.getNumObjetos());
23
24     Objeto uno,dos;
25     uno= new Objeto("");
26     dos= new Objeto("Objeto dos");
27     writeln("Objetos: ", uno.getNumObjetos());
28     writeln("Objetos: ", dos.getNumObjetos());
29     return 0;
30 }
```

Listing 137. Ejemplo de miembros estáticos en D.

Otro ejemplo:

```
1 module Persona;
2
```

```
3  import std.stdio;
4
5  class Persona{
6      private:
7          static int nPersonas;
8          static const int MAX=10;
9          string nombre;
10
11     public:
12         this(string s=null){
13             if(s!=null){
14                 nombre= s;
15                 writeln("Persona: ", nombre);
16             }else{
17                 nombre=null ;
18                 writeln("Persona: ");
19             }
20             nPersonas++;
21         }
22
23         ~this(){
24             writeln("eliminando persona : ", nombre);
25             nPersonas--;
26         }
27
28         static int getMax(){
29             return MAX;
30         }
31
32         static int getnPersonas(){
33             return nPersonas;
34         }
35     }
36
37     int main() {
38         writeln("Máximo de personas: ", Persona.getMax());
39         writeln("Número de personas: ", Persona.getnPersonas());
40
41         Persona perl= new Persona();
42         writeln("Máximo de personas: ", Persona.getMax());
43         writeln("Número de personas: ", Persona.getnPersonas());
44     }
```

```
45     Persona per2= new Persona("persona 2");  
46     writeln("Máximo de personas: ", per2.getMax());  
47     writeln("Número de personas: ", per2.getnPersonas());  
48     return 0;  
49 }
```

Listing 138. Ejemplo 2 de miembros estáticos en D.
Como en Java, no estamos obligados a inicializar los atributos estáticos.

Capítulo 15

Objetos constantes

Algunas ocasiones puede ser útil tener objetos constantes, los cuales no puedan ser modificados. Sin embargo, cada lenguaje interpreta de manera ligeramente distinta, como se verá a continuación.

15.1 Objetos constantes en C++

En C++, es posible tener objetos de tipo constante, los cuales no podrán ser modificados en ningún momento¹ Tratar de modificar un objeto constante se detecta como un error en tiempo de compilación.

Sintaxis
<code>const <clase> <lista de objetos>;</code>

Ejemplo:

```
const Hora h1(9, 30, 20);
```

Para estos objetos, algunos compiladores llegan a ser tan rígidos en el cumplimiento de la instrucción, que no permiten que se hagan llamadas a métodos sobre esos objetos. La compilación estándar permite la ejecución de métodos, siempre y cuando no modifiquen el estado del objeto.

Si se quiere consultar al objeto mediante llamadas a métodos `get`, lo correcto es declarar métodos con la palabra reservada `const`, para permitirles actuar libremente sobre los objetos sin modificarlo. La sintaxis requiere añadir después de la lista de parámetros la palabra reservada `const` en la declaración y en su definición.

¹ Ayuda a cumplir el principio del mínimo privilegio, donde se debe restringir al máximo el acceso a los datos cuando este acceso estaría de sobra[?].

Sintaxis
Declaración.
<code><tipo> <nombre> (<parámetros>) const;</code>
Definición del método fuera de la declaración de la clase.
<code><tipo> <clase> :: <nombre> (<parámetros>) const { <código> }</code>
Definición del método dentro de la declaración de la clase.
<code><tipo> <nombre> (<parámetros>) const { <código> }</code>

Los compiladores generalmente restringen el uso de métodos constantes a objetos constantes. Para solucionarlo es posible sobrecargar el método con la única diferencia de la palabra *const*, aunque el resto de la firma del método sea la misma.

Un método puede ser declarado dos veces tan sólo con que la firma del método difiera por el uso de *const*. Objetos constantes ejecutarán al método definido con *const*, y objetos variables ejecutarán al método sin esta restricción. De hecho, un objeto variable puede ejecutar el método no definido con *const* por lo que si el objetivo del método es el mismo, y este no modifica al objeto (e.g., métodos tipo *get*) bastaría con definir al método una vez².

Los constructores no necesitan la declaración *const*, puesto que deben poder modificar al objeto.

Ejemplo:

```

1  #include <iostream>
2  #include <time.h>
3  #include <stdlib.h>
4
```

²Además, declarar a los métodos *get* y otros métodos que no modifican al objeto con el calificador *const* es una buena práctica de programación.

```
5 using namespace std;
6
7 class Arr{
8     private:
9         int a[10];
10    public:
11        Arr(int x=0) {
12            srand( time(nullptr) );
13            for( int i=0; i<10; i++){
14                if (x==0)
15                    x=rand() %100;
16                a[i]=x;
17            }
18        }
19        char set(int, int);
20        int get(int) const ;
21        int get(int);
22    };
23
24    char Arr::set(int pos, int val ){
25        if(pos>=0 && pos<10){
26            a[pos]=val;
27            return 1;
28        }
29        return 0;
30    }
31
32    int Arr::get(int pos) const {
33        if(pos>=0 && pos<10)
34            return a[pos];
35        // a[9]=0; error en un método constante
36        return 0;
37    }
38
39    int Arr::get(int pos) { //no es necesario sobrecargar
40        if(pos>=0 && pos<10) // si el método no modifica
41            return a[pos];
42        return 0;
43    }
44
45    int main(){
46        const Arr a(5),b;
```

```

47     Arr c;
48
49     //      a.set(0,1); //error llamar a un método no const
50     //      b.set(0,2);          // para un objeto constante (comentar estas lineas)
51     c.set(0,3);
52     //      a.set(1,11); //error llamar a un método no const
53     //      b.set(1,22); // para un objeto constante (comentar estas lineas)
54     c.set(1,33);
55     cout<<a.get(0)<<endl; // ejecuta int get(int) const ;
56     cout<<a.get(1)<<endl;
57     cout<<b.get(0)<<endl;
58     cout<<b.get(1)<<endl;
59     cout<<c.get(0)<<endl; // ejecuta int get(int);
60     cout<<c.get(1)<<endl;
61     return 0;
62 }

```

Listing 139. Ejemplo de objetos constantes en C++.

15.2 Objetos finales en Java

Ya se mencionó en la sección de fundamentos de Java el uso de la palabra reservada *final*, la cual permite a una variable ser inicializada sólo una vez. En el caso de los objetos o referencias a los objetos el comportamiento es el mismo. Si se agrega la palabra *final* a la declaración de una referencia a un objeto, significa que la variable podrá ser inicializada una sola vez, en el momento que sea necesario.

Sintaxis

```
final <clase> <lista de identificadores de objetos>;
```

Por ejemplo:

```
final Hora h1= new Hora(9,30,20);
```

Concepto

Es importante remarcar que no es el mismo sentido de *const* en C++. Aquí lo único que se limita es la posibilidad de una variable de referencia a ser inicializada de nuevo, pero no inhibe la modificación de miembros.

Por ejemplo:

```
1 final Light aLight = new Light(); // variable local final
2 aLight.noOfWhatts = 100;           //Ok. Cambio en el edo. del objeto
3
4 aLight = new Light();              // Inválido. No se puede modificar la referencia
```

Ejemplo:

```
1 class Fecha {
2     private int dia;
3     private int mes, año;
4
5     public Fecha() {
6         dia=mes=1;
7         año=1900;
8     }
9     public boolean setDia(int d) {
10         if (d >=1 && d<=31) {
11             dia= d;
12             return true;
13         }
14         return false;
15     }
16     //poner día
17     public int getDia() {
18         return dia;
19     } //devuelve día
20     public boolean setMes(int m) {
21         if (m>=1 && m<=12) {
22             mes=m;
23             return true;
24         }
25         return false;
26     }
27     public int getMes() {
28         return mes;
29     }
30
31     public boolean setAño(int a) {
32         if (a>=1900) {
```

```
33         año=a;
34         return true;
35     }
36     return false;
37 }
38 public int getAño() {
39     return año;
40 }
41 }
42
43 public class MainF {
44
45     public static void main(String[] args) {
46         final Fecha f;
47
48         f= new Fecha();
49
50         f.setDia(10);
51         f.setMes(3);
52         f.setAño(2001);
53         System.out.println(f.getDia()+"/"+f.getMes()+"/"+f.getAño());
54         f= new Fecha(); //Error: la variable f es final y no puede ser reasigna
55     }
56 }
```

Listing 140. Ejemplo de objetos finales en Java.

15.3 Objetos constantes en Ruby

Los objetos constantes ya fueron indirectamente explicados al mencionar el manejo de constantes en Ruby, debido a que todos los datos son manejados como objetos. Recordemos que deben empezar con una **letra mayúscula** y pueden ser definidas dentro de una clase o módulo y serán visibles dentro de ese ámbito. Una constante definida fuera de un clase o módulo será vista globalmente. Es posible reasignar un valor a una constante, pero esto producirá una advertencia (pero no un error).

Ejemplo:

```
1 class Fecha
2     def initialize
3         @dia= @mes =1
4         @anio=1900
5     end
6
7     def setDia dia
8         if dia>=1 && dia<=31
9             @dia=dia
10            return true
11        end
12        return false
13    end
14
15    def setMes mes
16        if mes>=1 && mes<=12
17            @mes=mes
18            return true
19        end
20        return false
21    end
22
23    def setAnio a
24        if a>=1900
25            @anio=a
26            return true
27        end
28        return false
29    end
30
31    attr_reader :dia, :mes, :anio
```

```
32 end
33
34 #Código de prueba
35
36 Fe = Fecha.new # es constante por ser nombrada con una mayúscula como prefijo
37
38 Fe.setDia 10
39 Fe.setMes 3
40 Fe.setAnio 2008
41
42 puts Fe.dia.to_s+"/"+Fe.mes.to_s+"/"+Fe.anio.to_s
43 # ó:
44 puts "#{Fe.dia}/#{Fe.mes}/#{Fe.anio}"
45
46 Fe=Fecha.new # genera un warning pero no es un error al ser reasignado el identificador
```

Listing 141. Ejemplo de objetos constantes en Ruby.

Tenemos entonces que los objetos constantes tienen una mayor similitud con Java, siendo la diferencia principal que la modificación de un objeto constante en Ruby únicamente genera un *warning*, mientras que en Java genera un error. Siendo Ruby un lenguaje interpretado es una buena solución que, al no poder ser detectado este problema en tiempo de compilación, solo genera una advertencia y continúe la ejecución.

15.4 Objetos constantes en Python

Python no cuenta con objetos ni variables constantes. Se acostumbra definir variables de referencia a objetos que no se deben cambiar con mayúsculas, pero no existe ningún mecanismo que el lenguaje proporciona para evitar la modificación de los objetos.

15.5 Objetos constantes en Scala

Scala cuenta con identificadores constantes definidos mediante *val* que tienen un comportamiento similar a los definidos con *final* en Java. Del mismo modo, puede agregarse el modificador *final* aunque el comportamiento es idéntico. Las diferencias básicas con el *final* de Java es que un identificador *val* debe ser inicializado al declararse y el identificador puede ser sobrescrito.

Ejemplo:

```
1 class Fecha {
2     var dia, mes: Int=0
3     var anio: Int=1900
4
5     def setDia(mdia: Int): Boolean ={
6         if (mdia>=1 && mdia<=31) {
7             dia=mdia
8             return true
9         }
10        return false
11    }
12
13    def setMes(mmes: Int): Boolean ={
14        if (mmes>=1 && mmes<=12) {
15            mes=mmes
16            return true
17        }
18        return false
19    }
20
21    def setAnio (manio: Int): Boolean ={
22        if (manio>=1900) {
23            anio=manio
24            return true
25        }
26        return false
27    }
28 }
29
30 //Código de prueba
31 object Constantes {
32     val fe = new Fecha() // similar a final en java, salvo que puede ser sobrescrito
33     final val fe2= new Fecha() // similar a final en java, salvo que no puede ser sobreescrito
```

```
34
35  def main(args: Array[String]) = {
36      System.out.println("Constantes")
37      var f= new Fecha()
38
39      f.setDia(10)
40      f.setMes(3)
41      f.setAnio(2010)
42
43      // fe2=fe //no se puede reasignar
44      // fe=fe2 //no se puede reasignar
45
46      val fe = new Fecha() // se puede sobrescribir fe
47      val fe2 = new Fecha() // se puede sobrescribir fe2
48      System.out.println("fin de Constantes")
49  }
50 }
```

Listing 142. Ejemplo de objetos constantes en Scala.

15.6 Objetos constantes en C#

C# no permite el uso de objetos constantes. Aunque existe la palabra clave *const*, ésta es usada para definir un identificador para una expresión constante. Esto no permite la asignación de objetos puesto que son creados dinámicamente. De hecho:

```
public const int i = x;
```

es un error si x no es constante.

15.7 Objetos constantes en D

El manejo de objetos constantes en D es básicamente como el de C++. Aparte de los ajustes en la sintaxis, la única diferencia es el uso de objetos dinámicos ya que D no maneja objetos estáticos.

```
1 module Arr;
2
3 import std.stdio;
4
5 class Arr{
6     private:
7     int a[10];
8     public:
9     this(int x=0) {
10         for( int i=0; i<10; i++){
11             if (x==0)
12                 x=i;
13             a[i]=x;
14         }
15     }
16
17     char set(int pos, int val ){
18         if(pos>=0 && pos<10){
19             a[pos]=val;
20             return 1;
21         }
22         return 0;
23     }
24
25     int get(int pos) const {
26         writeln("Ejecuta int get(int) const");
27         if(pos>=0 && pos<10)
28             return a[pos];
29         // a[9]=0; // error en un método constante
30         return 0;
31     }
32
33     int get(int pos) { //no es necesario sobrecargar
34         writeln("Ejecuta int get(int)");
35         if(pos>=0 && pos<10) // si el método no modifica
36             return a[pos];
```

```
37         return 0;
38     }
39 }
40 int main() {
41     const Arr a= new Arr(5), b=new Arr();
42     Arr c;
43
44     c = new Arr();
45
46     //     a.set(0,1); //error llamar a un método no const
47     //     b.set(0,2); // para un objeto constante (comentar estas lineas)
48     c.set(0,3);
49     //     a.set(1,11); //error llamar a un método no const
50     //     b.set(1,22); // para un objeto constante (comentar estas lineas)
51     c.set(1,33);
52     writeln(a.get(0)); // ejecuta int get(int) const ;
53     writeln(a.get(1));
54     writeln(b.get(0));
55     writeln(b.get(1));
56     writeln(c.get(0)); // ejecuta int get(int);
57     writeln(c.get(1));
58     return 0;
59 }
```

Listing 143. Ejemplo de objetos constantes en D.

Capítulo 16

Amistad en C++

En C++ existe el concepto de **amistad**. Aunque puede ser considerado por algunos como una intrusión a la encapsulación o a la privacidad de los datos:

Concepto
<i>"... la amistad corrompe el ocultamiento de información y debilita el valor del enfoque de diseño orientado a objetos"[?]</i>

Un amigo de una clase es una función u otra clase que no es miembro de la clase, pero que tiene permiso de usar los miembros públicos y privados de la clase¹.

Es importante señalar que el ámbito de una función amiga no es el de la clase, y por lo tanto los amigos no son llamados con los operadores de acceso de miembros.

¹También tiene acceso a los miembros protegidos que se verán más adelante.

Sintaxis

Sintaxis para una función amiga:

```
class <nombreClase> {  
    friend <tipo> <metodo>();  
    ...  
public:  
    ...  
};
```

Sintaxis para una clase amiga:

```
class <nombreClase> {  
    friend <nombreClaseAmiga>;  
    ...  
public:  
    ...  
};
```

Las funciones o clases amigas no son privadas ni públicas (o protegidas), pueden ser colocadas en cualquier parte de la definición de la clase, pero se acostumbra que sea al principio.[Deitel, 1995]

Como la amistad entre personas, esta es **concedida** y no tomada. Si la clase B quiere ser amigo de la clase A, la clase A debe declarar que la clase B es su amiga.

La amistad **no es simétrica ni transitiva**: si la clase A es un amigo de la clase B, y la clase B es un amigo de la clase C, no implica:

- Que la clase B sea un amigo de la clase A.
- Que la clase C sea un amigo de la clase B.
- Que la clase A sea un amigo de la clase C.

El concepto de amistad no está implementado en otros lenguajes aunque el nivel protegido permite un cierto nivel de acceso miembros de clases del mismo módulo en algunos lenguajes.

Ejemplo:

```

1  //Ejemplo de funcion amiga con acceso a miembros privados
2  #include <iostream>
3
4  using namespace std;
5
6  class ClaseX{
7      friend void setX(ClaseX &, int);  //declaración friend
8      public:
9          ClaseX() {
10              x=0;
11          }
12          void print() const {
13              cout<<x<<endl;
14          }
15          private:
16          int x;
17      };
18
19  void setX(ClaseX &c, int val){
20      c.x=val;          //es legal el acceso a miebros privados por amistad.
21  }
22
23  int main(){
24      ClaseX pr;
25
26      cout<<"pr.x después de instanciación : ";
27      pr.print();
28      cout<<"pr.x después de la llamada a la función amiga setX : ";
29      setX(pr, 10);
30      pr.print();
31  }

```

Listing 144. Ejemplo de funciones amigas en C++.

```

1  //ejemplo 2 de funciones amigas
2  #include <iostream>
3  using namespace std;
4
5  class Linea;
6
7  class Recuadro {

```

```
8      friend int mismoColor(Linea, Recuadro);
9
10     private:
11     int color; //color del recuadro
12     int xsup, ysup; //esquina superior izquierda
13     int xinf, yinf; //esquina inferior derecha
14
15     public:
16     void ponColor(int);
17     void definirRecuadro(int, int, int, int);
18 };
19
20 class Linea{
21     friend int mismoColor(Linea, Recuadro);
22
23     private:
24     int color;
25     int xInicial, yInicial;
26     int lon;
27
28     public:
29     void ponColor(int);
30     void definirLinea(int, int, int);
31 };
32
33 int mismoColor(Linea l, Recuadro r){
34     if(l.color==r.color)
35         return 1;
36     return 0;
37 }
38
39 //métodos de la clase Recuadro
40 void Recuadro::ponColor(int c) {
41     color=c;
42 }
43
44 void Recuadro::definirRecuadro(int x1, int y1, int x2, int y2) {
45     xsup=x1;
46     ysup=y1;
47     xinf=x2;
48     yinf=y2;
49 }
```

```
50
51 //métodos de la clase Linea
52 void Linea::ponColor(int c) {
53     color=c;
54 }
55
56 void Linea::definirLinea(int x, int y, int l) {
57     xInicial=x;
58     yInicial=y;
59     lon=l;
60 }
61
62 int main() {
63     Recuadro r;
64     Linea l;
65
66     r.definirRecuadro(10, 10, 15, 15);
67     r.ponColor(3);
68     l.definirLinea(2, 2, 10);
69     l.ponColor(4);
70     if(!mismoColor(l, r))
71         cout<<"No tienen el mismo color"<<endl;
72     //se ponen en el mismo color
73     l.ponColor(3);
74     if(mismoColor(l, r))
75         cout<<"Tienen el mismo color";
76     return 0;
77 }
```

Listing 145. Ejemplo 2 de funciones amigas en C++.

Capítulo 17

Polimorfismo AdHoc: Sobrecarga de operadores

La sobrecarga de operadores es la capacidad de definir nuevo comportamiento para operadores existentes en un lenguaje con tipos de datos definidos por el usuario. De esta forma, en programación orientada a objetos, se les da a los operadores un nuevo significado de acuerdo al objeto sobre el cual se aplique.

C++, Ruby, Scala¹, C#² y D³ permiten la sobrecarga de operadores. Java no permite la sobrecarga de operadores. Python cuenta con una aproximación a la sobrecarga de operadores.

Concepto
Para sobrecargar un operador, se define un método que es invocado cuando el operador es aplicado sobre ciertos tipos de datos.

¹[Scala: Overloading Operators](#)

²[Operator Overloading Tutorial](#)

³[Operator Overloading](#)

17.1 Sobrecarga de operadores en C++

La sintaxis para definir un método con un operador difiere de la definición normal de un método, pues el nombre del método está definido por la palabra reservada *operator* y el operador que se va a sobrecargar:

Sintaxis
<pre><tipo> operator <operador> (<argumentos>) ;</pre>
o
<pre><tipo> operator <operador> (<argumentos>) { <cuerpo del método> }</pre>
Para la definición fuera de la clase:
<pre><tipo> <clase>::operator <operador> (<argumentos>) { <cuerpo del método> }</pre>

Para utilizar un operador con objetos, es necesario que el operador esté sobrecargado, aunque existen dos excepciones:

- El **operador de asignación** `=`, puede ser utilizado sin sobrecargarse explícitamente, pues el comportamiento por omisión es una **copia a nivel de miembro** de los miembros de la clase. Sin embargo no debe de usarse si la clase cuenta con miembros a los que se les asigne memoria de manera dinámica.
- El **operador de dirección** `&`, está sobrecargado por omisión para devolver la dirección de un objeto de cualquier clase.

Algunas restricciones:

1. Operadores que no pueden ser sobrecargados: `..*` `::?` `:` `sizeof`
2. La precedencia de un operador no puede ser modificada. Deben usarse los paréntesis para obligar un nuevo orden de evaluación.

3. La asociatividad de un operador no puede ser modificada.
4. No se puede modificar el número de operandos de un operador. Los operadores siguen siendo unarios o binarios.
5. No es posible crear nuevos operadores.
6. No puede modificarse el comportamiento de un operador sobre tipos de datos definidos por el lenguaje.

Ejemplo:

```
1  //programa de ejemplo de sobrecarga de operadores.
2  class Punto {
3      float x, y;
4  public:
5      Punto(float xx=0, float yy=0) {
6          x=xx;
7          y=yy;
8      }
9      Punto operator =(Punto);
10     Punto operator +(Punto);
11     Punto operator -();
12     Punto operator *(float);
13     Punto operator *=(float);
14     Punto operator ++(); //prefijo
15     Punto operator ++(int); //posfijo
16     bool operator >(Punto);
17     bool operator <=(Punto);
18 };
19
20 Punto Punto::operator =(Punto a) { //copia o asignación
21     x=a.x;
22     y=a.y;
23     return *this;
24 }
25
26 Punto Punto::operator +(Punto p) {
27     return Punto(x+p.x, y+p.y);
28 }
29
30 Punto Punto::operator -() {
31     return Punto(-x, -y);
```

```
32 }
33
34 Punto Punto::operator * (float f) {
35     Punto temp;
36     temp=Punto(x*f, y*f);
37     return temp;
38 }
39
40 // incremento prefijo
41 Punto Punto::operator ++() {
42     x++;
43     y++;
44     return *this;
45 }
46
47 // incremento posfijo
48 Punto Punto::operator ++(int)
49 {
50     Punto temp= *this;
51     x++;
52     y++;
53     return temp;
54 }
55
56 bool Punto::operator > (Punto p) {
57     return (x>p.x && y>p.y) ? 1 : 0;
58 }
59
60 bool Punto::operator <= (Punto p) {
61     return (x<=p.x && y<=p.y) ? 1 : 0;
62 }
63
64 int main() {
65     Punto a(1,1);
66     Punto b;
67     Punto c;
68
69     b++;
70     ++b;
71     c=b;
72     c=a+b;
73     c=-a;
```

```
74         c=a*.5;  
75  
76         return 0;  
77     }
```

Listing 146. Ejemplo de sobrecarga de operadores con C++.

Ejercicio

Crear una clase *String* para el manejo de cadenas. Tendrá dos atributos: apuntador a carácter y un entero *tam*, para almacenar el tamaño de la cadena. Sobrecargar operadores = (asignación), (==) igualdad, !=, <, >, <=, >=, +, + =, []. Generar un programa de prueba para objetos de la clase. La estructura inicial será la siguiente:

```
class String{  
    char *s;  
    int tam;  
public:  
    String(char * =nullptr);  
    String(const String &copia); //constructor de copia  
    ~String();  
    //sobrecarga de operador de asignación  
    const String &operator =(const String &);  
    //igualdad  
    bool operator ==(const String &) const ;  
};
```

Ejemplo: código de *String*

Véase que es posible asignar una cadena sin sobrecargar el operador de asignación, o comparar un objeto *String* con una cadena. Esto se logra gracias a que se provee de un constructor que convierte una cadena a un objeto *String*. De esta manera, este **constructor de conversión** es llamado automáticamente, creando un objeto temporal para ser comparado con el otro objeto.

No es posible que la cadena (o apuntador a char) vaya del lado izquierdo, pues se estaría llamando a la funcionalidad del operador para un apuntador a char. Si se requiere que el operando izquierdo sea de un tipo distinto al de la clase, la sobrecarga del operador debe hacerse usando una función externa a la clase. Si fuera necesario, esta función podría declararse como amiga de la clase.

```
1  //Sobrecarga de operadores. Implementación de una clase String
2  #include <iostream>
3  #include <stdio.h>
4  #include <string.h>
5
6  using namespace std;
7
8  class String{
9      //operadores de salida de flujo
10     friend ostream &operator << (ostream &, const String &);
11 private:
12     char *s;
13     int tam;
14 public:
15     String(const char * =nullptr);
16
17     String(const String &copia){
18         s=nullptr;
19         tam=0;
20         *this=copia;//¿se vale o no?
21     }
22     ~String(){
23         if(s!=nullptr){
24             delete []s;
25             s=nullptr;
26             tam=0;
27         }
28     }
29
30     //sobrecarga de constructor de asignación
```

```

31     const String &operator =(const String &);
32
33     //igualdad
34     bool operator ==(const String &) const ;
35
36     //concatenación
37     String operator +(const String &) const;
38
39     //concatenación y asignación
40     const String &operator +=(const String &);
41
42     String &copiar (const String &);
43
44     //sobrecarga de los corchetes
45     char &operator[] (int) const;
46 };
47
48 //operadores de inserción y extracción de flujo
49 ostream& operator<< (ostream &salida, const String &cad){
50     salida<<cad.s;
51     return salida; //permite concatenación
52 }
53
54 istream &operator >> (istream &entrada, String &cad){
55     char tmp[100];
56     entrada >> tmp;
57     cad=tmp; //usa operador de asignación de String y const. de conversión
58     return entrada; //permite concatenación
59 }
60
61 String::String(const char *c){
62     if(c==nullptr){
63         s=nullptr;
64         tam=0;
65     } else {
66         tam=strlen(c);
67         s= new char[tam+1];
68         strcpy(s, c);
69     }
70 }
71
72 const String &String::operator =(const String &c){

```

```
73         if(this!= &c) {           //verifica no asignarse a si mismo
74             if(s!=nullptr)
75                 delete []s;
76             tam=c.tam;
77             s= new char[tam+1];
78             strcpy(s, c.s);
79         }
80         return *this; //permite concatenación de asignaciones
81     }
82
83     bool String::operator ==(const String &c) const {
84         return strcmp(s, c.s)==0;
85     }
86
87     //operador de suma regresa una copia de la suma obtenida
88     //en un objeto local.
89     String String::operator +(const String &c) const {
90         String tmp(*this);
91         tmp+=c;
92         return tmp;
93     }
94
95     const String &String::operator +=(const String &c){
96         char *str=s, *ctmp= new char [c.tam+1];
97         strcpy(ctmp, c.s);
98         tam+=c.tam;
99         s= new char[tam+1];
100        strcpy(s, str);
101        strcat(s, ctmp);
102        delete []str;
103        delete []ctmp;
104        return *this;
105    }
106
107    String &String::copia (const String &c) {
108        if(this!= &c) {           //verifica no asignarse a si mismo
109            if(s!=nullptr)
110                delete []s;
111            tam=c.tam;
112            s= new char[tam+1];
113            strcpy(s, c.s);
114        }
```



```
115         return *this; //permite concatenación de asignaciones
116     }
117
118     char &String::operator[] (int i) const {
119         if(i>=0 && i<tam)
120             return s[i];
121         return s[0];
122     }
123
124     int main(){
125         String a("AAA");
126         String b("Prueba de cadena");
127         String c(b);
128         // Es un error hacer una asignación sin liberar memoria.
129         // ese es el principal peligro de usar el operador sobrecargado
130         // por default de asignación
131         a=b;
132         b.copia("H o l a");
133         b=c+c;
134         b="nueva";
135         c+=c;
136         String d("nueva cadena");
137         d+="Hola";
138         String e;
139         e=d+"Adios";
140         d="coche";
141         int x=0;
142         x=d=="coche"; //Lo contrario no es válido "coche"==d
143         char ch;
144         ch=d[7];
145         d[2]='X';
146         cout<<d<<endl;
147         cout<<"Introduce dos cadenas:";
148         cin>>e>>d;
149         cout<<"Cadenas:\n";
150         cout<<e<<endl<<d;
151         return 0;
152     }
```

Listing 147. Ejemplo de String con sobrecarga de operadores en C++.

17.2 Sobrecarga de operadores en Ruby

Aunque Ruby no maneja sobrecarga de operaciones si soporta la sobrecarga de operadores. Para hacerlo simplemente se define un método en la clase, donde el nombre del método es el nombre del operador que se necesita sobrecargar.

Sintaxis

```
def <operador>
  ...
end
```

Operadores que no pueden ser sobrecargados:

`=, !, not, &&, and,`

`!, or, ., !=, .., ..., ::` and some others ¿- ?. |

La instrucción *return* puede omitirse en Ruby porque regresa el resultado de la ultima expresión de una función.

Ejemplo:

```
1  #programa de ejemplo de sobrecarga de operadores.
2  class Punto
3    attr_reader :x, :y
4    @x
5    @y
6    def initialize x=10, y=10
7      @x, @y=x,y
8    end
9
10   def + op
11     return Punto.new(@x+op.x, @y+op.y)
12   end
13
14   def - (op)
15     Punto.new(@x-op.x, @y-op.y)
16   end
17
```

```
18  def * (f)
19      temp=Punto.new @x*f, @y*f
20      return temp
21  end
22
23  def > (op)
24      return (@x>op.x && @y>op.y) ? true : false
25  end
26
27  def <= (op)
28      return (@x<=op.x && @y<=op.y) ? true : false
29  end
30  end
31
32  def prueba
33      a= Punto.new 1,1
34      b= Punto.new
35      c= Punto.new
36      puts a.x
37      puts b.x
38      puts c.x
39
40      c=a+b
41      puts c.x
42      puts c.y
43      c+=a
44      puts c.x
45      puts c.y
46      c=a*0.5
47      puts c.x
48      puts c.y
49      p=a-b
50      puts p.x
51      puts p.y
52
53      puts a>b
54      puts a<=b
55  end
56
57  prueba
```

Listing 148. Ejemplo de sobrecarga en Ruby.

Al ser un lenguaje dinámico, el segundo operando queda determinado en tiempo de ejecución. Además al sobrecargar, por ejemplo, el operador `+` automáticamente queda sobrecargado el operador `+=` (lo mismo para el resto de las asignaciones abreviadas).

17.3 Sobrecarga de operadores en Python

Python permite la sobrecarga de operadores mediante la definición de métodos especiales⁴

Capítulo 18

Herencia

18.1 Introducción

La **herencia** es un mecanismo potente de abstracción que permite compartir similitudes entre clases manteniendo al mismo tiempo sus diferencias.

Es una forma de reutilización de código, tomando clases previamente creadas y formando a partir de ellas nuevas clases, heredándoles sus atributos y métodos. Las nuevas clases pueden ser modificadas agregándoles nuevas características.

Los términos para distinguir los tipos de clases pueden variar. Por ejemplo, en C++ la clase de la cual se toman sus características se conoce como **clase base**; mientras que la clase que ha sido creada a partir de la clase base se conoce como **clase derivada**. Existen otros términos para estas clases:

En Java es más común usar el término de superclase y subclase.

Una clase derivada es potencialmente una clase base, en caso de ser necesario.

Cada objeto de una clase derivada también es un objeto de la clase base. En cambio, un objeto de la clase base no es un objeto de la clase derivada.

La implementación de herencia a varios niveles forma un árbol jerárquico similar al de un árbol genealógico. Esta es conocida como **jerarquía de herencia**.

Generalización. Una clase base o superclase se dice que es más general que la clase derivada o subclase.

Especialización. Una clase derivada es por naturaleza una clase más especializada que su clase base.

Clase base	Clase derivada
Superclase	Subclase
Clase padre	Clase hija

18.2 Herencia: Implementación en C++

La herencia en C++ es implementada permitiendo a una clase incorporar a otra clase dentro de su declaración.

Sintaxis

```
class <claseDerivada>: <acceso> <claseBase> {  
    //cuerpo clase derivada  
};
```

Ejemplo: Una clase *vehículo* que describe a todos aquellos objetos vehículos que viajan en carreteras. Puede describirse a partir del número de ruedas y de pasajeros. De la definición de vehículos podemos definir objetos más específicos (especializados). Por ejemplo la clase *camión*.

```
1  #include <iostream>  
2  using namespace std;  
3  
4  class Vehiculo{  
5      int ruedas;  
6      int pasajeros;  
7  public:  
8      void setRuedas (int) ;  
9      int  getRuedas () ;  
10     void setPasajeros (int) ;  
11     int  getPasajeros () ;  
12 };  
13  
14 void Vehiculo::setRuedas (int num) {  
15     ruedas=num;  
16 }  
17 int Vehiculo::getRuedas () {  
18     return ruedas;  
19 }  
20 void Vehiculo::setPasajeros (int num) {  
21     pasajeros=num;  
22 }
```



```
23
24 int Vehiculo::getPasajeros() {
25     return pasajeros;
26 }
27
28 //clase Camion con herencia de Vehículo
29 class Camion: public Vehiculo {
30     int carga;
31     public:
32     void setCarga(int);
33     int getCarga();
34     void muestra();
35 };
36
37 void Camion::setCarga(int num) {
38     carga=num;
39 }
40 int Camion::getCarga() {
41     return carga;
42 }
43 void Camion::muestra() {
44     cout<<"Ruedas: "<< getRuedas() <<endl;
45     cout<<"Pasajeros: "<< getPasajeros() <<endl;
46     cout<<"Capacidad de carga: "<<getCarga() <<endl;
47 }
48
49 int main() {
50     Camion ford;
51     ford.setRuedas(6);
52     ford.setPasajeros(3);
53     ford.setCarga(3200);
54     ford.muestra();
55     return 0;
56 }
```

Listing 149. Ejemplo de herencia en C++, jerarquía de Vehículo.

18.2.1 Control de Acceso a miembros en C++

Existen tres palabras reservadas para el control de acceso: *public*, *private* y *protected*. Estas sirven para proteger los miembros de la clase en diferentes formas.

El control de acceso, como ya se vio anteriormente, se aplica a los métodos,

atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

- **private.** Un miembro privado únicamente puede ser utilizado por los métodos miembro y funciones amigas de la clase donde fue declarado.
- **protected.** Un miembro protegido puede ser utilizado únicamente por los métodos miembro y funciones amigas de la clase donde fue declarado o por los métodos miembro y funciones amigas de las clases derivadas. El acceso protegido es como un nivel intermedio entre el acceso privado y público.
- **public.** Un miembro público puede ser utilizado por cualquier método. Una estructura es considerada por C++ como una clase que tiene todos sus miembros públicos.

```
1 //ejemplo de control de acceso
2
3 class S{
4     char *f1 ();
5     int a;
6     protected:
7     int b;
8     int f2 ();
9     private:
10    int c;
11    int f3 ();
12    public:
13    int d, f;
14    char *f4 (int);
15 };
16
17 int main() {
18     S obj;
19     obj.f1 (); //error
20     obj.a=1; //error
21     obj.f2 (); //error
22     obj.b=2; //error
23     obj.c=3; //error
24     obj.f3 (); //error
25     obj.d=5;
26     obj.f4 (obj.f);
```

```
27     return 0;  
28 }
```

Listing 150. Ejemplo de herencia y control de acceso a miembros en C++.

18.2.2 Control de acceso en herencia en C++

Hasta ahora se ha usado la herencia con un solo tipo de acceso, utilizando el especificador *public*. Los miembros públicos de la clase base son miembros públicos de la clase derivada, los miembros protegidos permanecen protegidos para la clase derivada.

Ejemplo:

```
1  //ejemplo de control de acceso en herencia  
2  
3  class Base{  
4      int a;  
5      protected:  
6      int b;  
7      public:  
8      int c;  
9  };  
10  
11  class Derivada: public Base {  
12      void g();  
13  };  
14  
15  void Derivada::g() {  
16      a=0; //error, es privado  
17      b=1; //correcto, es protegido  
18      c=2; //correcto, es público  
19  }
```

Listing 151. Ejemplo control de acceso en herencia en C++.

Para acceder a los miembros de una clase base desde una clase derivada, se pueden ajustar los permisos por medio de un calificador *public*, *private* o *protected*.

Si una clase base es declarada como **pública** de una clase derivada, los miembros públicos y protegidos son accesibles desde la clase derivada, no así los miembros privados.

Si una clase base es declarada como **privada** de otra clase derivada, los miembros públicos y protegidos de la clase base serán miembros privados de la clase

derivada. Los miembros privados de la clase base permanecen inaccesibles.

Si se omite el calificador de acceso de una clase base, se asume **por omisión** que el calificador es *public* en el caso de una estructura y *private* en el caso de una clase.

Ejemplo de sintaxis:

```
1  class base {
2      ...
3  };
4
5  class d1: private base {
6      ...
7  };
8
9  class d2: base {
10     ...
11 };
12
13 class d3: public base {
14     ...
15 };
```

Es recomendable declarar explícitamente la palabra reservada *private* al tomar una clase base como privada para evitar confusiones:

```
1  class x{
2      public:
3      f();
4  };
5
6  class y: x {    //privado por omisión
7      ...
8  };
9
10 void g( y *p){
11     p->f();    //error
12 }
```

Finalmente, si una clase base es declarada como **protegida** de una clase derivada, los miembros públicos de la clase base se convierten en miembros protegidos de la clase derivada.

Ejemplo:

```
1 //acceso por herencia
2 #include <iostream>
3 using namespace std;
4
5 class X{
6     protected:
7         int i;
8         int j;
9
10    public:
11        void preg_ij();
12        void pon_ij();
13 };
14
15 void X::preg_ij() {
16     cout<< "Escriba dos números: ";
17     cin>>i>>j;
18 }
19
20 void X::pon_ij() {
21     cout<<i<<' '<<j<<endl;
22 }
23
24 //en Y, i y j de X siguen siendo miembros protegidos
25 //Si se llegara a cambiar este acceso a private i y j se heredarían como
26 // miembros privados de Y, además de los métodos públicos
27 class Y: public X{
28     int k;
29 public:
30     int preg_k();
31     void hacer_k();
32 };
33
34 int Y:: preg_k() {
35     return k;
36 }
37
38 void Y::hacer_k() {
39     k=i*j;
40 }
41
```

```
42 // Z tiene acceso a i y j de X, pero no a k de Y
43 // porque es private por omisión
44 // Si Y heredara de x como private, i y j serían privados en Y,
45 // por lo que no podrían ser accesados desde Z
46 class Z: public Y {
47 public:
48     void f();
49 };
50
51 // Si Y heredara a X con private, este método ya no funcionaría
52 // no se podría acceder a i ni a j.
53 void Z::f() {
54     i=2;
55     j=3;
56 }
57
58 // si Y hereda de x como private, no es posible acceder a los métodos
59 // públicos desde objetos de Y ni de Z.
60 int main() {
61     Y var;
62     Z var2;
63
64     var.preg_ij();
65     var.pon_ij();
66     var.hacer_k();
67     cout<<var.preg_k()<<endl;
68     var2.f();
69     var2.pon_ij();
70     return 0;
71 }
```

Listing 152. Ejemplo control de acceso en herencia con C++.

18.2.3 Constructores de clase base en C++

El constructor de la clase base puede ser llamado desde la clase derivada, para inicializar los atributos heredados. La sintaxis es igual que el inicializador de objetos componentes.

Los constructores y operadores de asignación de la clase base no son heredados por las clases derivadas. Pero pueden ser llamados por los de la clase derivada.

Un constructor de la clase derivada llama primero al constructor de la clase base. Si se omite el constructor de la clase derivada, el constructor por omisión de

la clase derivada llamará al constructor de la clase base.

Los destructores son llamados en orden inverso a las llamadas del constructor: un destructor de una clase derivada será llamado antes que el de su clase base.

Sintaxis

```
<clase>::<constructor>(<lista de argumentos>) :  
    <constructor de clase base>(<lista de argumentos sin el tipo>)
```

Ejemplo

```
1  // POINT.H  
2  #ifndef POINT_H  
3  #define POINT_H  
4  
5  class Point {  
6  public:  
7      Point(float = 0.0, float = 0.0);  
8      ~Point();  
9  protected:  
10     float x, y;  
11 };  
12  
13 #endif /*POINT_H*/  
14  
15 // POINT.CPP  
16 #include <iostream>  
17 #include "point.h"  
18 using namespace std;  
19 Point::Point(float a, float b){  
20     x = a;  
21     y = b;  
22  
23     cout << "Constructor Point: "  
24         << '[' << x << ", " << y << ']' << endl;  
25 }  
26  
27 Point::~~Point() {
```

```

28     cout << "Destructor Point:  "
29         << '[' << x << ", " << y << ']' << endl;
30 }
31
32 // CIRCLE.H
33 #ifndef CIRCLE_H
34 #define CIRCLE_H
35
36 #include "point.h"
37
38 class Circle : public Point {
39 public:
40     Circle(float r = 0.0, float x = 0, float y = 0);
41     ~Circle();
42 private:
43     float radius;
44 };
45
46 #endif /*CIRCLE_H*/
47
48 // CIRCLE.CPP
49 #include "circle.h"
50
51 using namespace std;
52
53 Circle::Circle(float r, float a, float b)
54     : Point(a, b)    // llamada al constructor de clase base
55 {
56     radius = r;
57
58     cout << "Constructor Circle: radio es "
59         << radius << " [" << a << ", " << b << ']' << endl;
60 }
61
62 Circle::~~Circle() {
63     cout << "Destructor Circle:  radio es "
64         << radius << " [" << x << ", " << y << ']' << endl;
65 }
66
67 // Main.CPP
68 #include <iostream.h>
69 #include "point.h"

```



```
70 #include "circle.h"
71
72 int main() {
73     // Muestra llamada a constructor y destructor de Point
74     {
75         Point p(1.1, 2.2);
76     }
77
78     cout << endl;
79     Circle circle1(4.5, 7.2, 2.9);
80     cout << endl;
81     Circle circle2(10, 5, 5);
82     cout << endl;
83     return 0;
84 }
```

Listing 153. Ejemplo constructor de clase base en C++.

18.2.4 Manejo de objetos de la clase base como objetos de una clase derivada y viceversa en C++

Un objeto de una clase derivada pública, puede ser manejado como un objeto de su clase base. Sin embargo, un objeto de la clase base no es posible tratarlo de forma automática como un objeto de clase derivada.

La opción que se puede utilizar, es enmascarar un objeto de una clase base a un apuntador de clase derivada. El problema es que no debe ser desreferenciado (accedido) así, primero se tiene que hacer que el objeto sea referenciado por un apuntador de su propia clase.

Si se realiza la conversión explícita de un apuntador de clase base - que apunta a un objeto de clase base - a un apuntador de clase derivada y, posteriormente, se hace referencia a miembros de la clase derivada, es un error pues esos miembros no existen en el objeto de la clase base.

Ejemplo:

```
1 // POINT.H
2 // clase Point
3 #ifndef POINT_H_
4 #define POINT_H_
5 using namespace std;
6
7 class Point {
```

```
8      friend ostream &operator<<(ostream &, const Point &);
9      public:
10     Point(float = 0, float = 0);
11     void setPoint(float, float);
12     float getX() const { return x; }
13     float getY() const { return y; }
14     protected:
15     float x, y;
16 };
17
18 #endif /*POINT_H*/
19
20
21 // POINT.CPP
22 #include <iostream>
23 #include "point.h"
24
25 Point::Point(float a, float b){
26     x = a;
27     y = b;
28 }
29
30 void Point::setPoint(float a, float b){
31     x = a;
32     y = b;
33 }
34
35 ostream &operator<<(ostream &output, const Point &p){
36     output << '[' << p.x << ", " << p.y << ']'<< '\n';
37
38     return output;
39 }
40
41
42 // CIRCLE.H
43 // clase Circle
44 #ifndef CIRCLE_H
45 #define CIRCLE_H
46
47 #include <iostream>
48 #include <iomanip>
49 #include "point.h"
```

```

50
51 class Circle : public Point { // Circle hereda de Point
52     friend ostream &operator<<(ostream &, const Circle &);
53     public:
54         Circle(float r = 0.0, float x = 0, float y = 0);
55
56         void setRadius(float);
57         float getRadius() const;
58         float area() const;
59     protected:
60         float radius;
61 };
62 #endif /*CIRCLE_H_*/
63
64
65 // CIRCLE.CPP
66 #include "circle.h"
67
68 Circle::Circle(float r, float a, float b)
69     : Point(a, b) // llama al constructor de la clase base
70 { radius = r; }
71
72 void Circle::setRadius(float r) { radius = r; }
73
74 float Circle::getRadius() const { return radius; }
75
76 float Circle::area() const
77 { return 3.14159 * radius * radius; }
78
79 // salida en el formato:
80 // Center = [x, y]; Radius = #.##
81 ostream &operator<<(ostream &output, const Circle &c){
82     output << "Center = [" << c.x << ", " << c.y
83         << "]; Radius = " << setprecision(2) << c.radius;
84
85     return output;
86 }
87
88 //Prueba.cpp
89 // Probando apunadores a clase base a apunadores a clase derivada
90 #include <iostream>
91 #include <iomanip>

```

```

92  #include "point.h"
93  #include "circle.h"
94
95  int main() {
96      Point *pointPtr, p(3.5, 5.3);
97      Circle *circlePtr, c(2.7, 1.2, 8.9);
98
99      cout << "Point p: " << p << "\nCircle c: " << c << endl;
100
101      // Maneja a un Circle como un Circle
102      pointPtr = &c;    // asigna la direccion de Circle a pointPtr
103      circlePtr = (Circle *) pointPtr; // mascara de base a derivada
104      cout << "\nArea de c (via circlePtr): "
105      << circlePtr->area() << endl;
106
107      // Es riesgoso manejar un Point como un Circle
108      // getRadius() regresa basura
109      pointPtr = &p;    // asigna direccion de Point a pointPtr
110      circlePtr = (Circle *) pointPtr; // mascara de base a derivada
111      cout << "\nRadio de objeto apuntado por circlePtr: "
112      << circlePtr->getRadius() << endl;
113
114      return 0;
115  }

```

Listing 154. Ejemplo de objetos de clase derivada como clase base en C++.

El operador *dynamic_cast*

El operador *dynamic_cast* es una forma de realizar una conversión de punteros o referencias de una clase base a una clase derivada en tiempo de ejecución. Este operador se utiliza para comprobar si un objeto de una clase base es realmente un objeto de una clase derivada específica ¹.

La sintaxis para utilizar el operador *dynamic_cast* es:

Sintaxis
<code>dynamic_cast<tipo_derivado>(puntero_o_referencia_base)</code> height

¹La contraparte es el operador *static_cast* es otra forma de realizar una conversión de apuntadores o referencias en C++. A diferencia del *dynamic_cast*, el *static_cast* realiza la conversión de forma estática en tiempo de compilación, es decir, no verifica si la conversión es válida.

Supongamos que tenemos una clase base *Empleado* y dos clases derivadas *EmpleadoPorHora* y *EmpleadoFijo*.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Empleado {
6  public:
7      virtual double salario() = 0;
8  };
9
10 class EmpleadoPorHora : public Empleado {
11     private:
12         double horasTrabajadas;
13         double tarifaPorHora;
14     public:
15         // constructor con inicialización directa de miembros previa a
16         // la ejecución del constructor
17         EmpleadoPorHora(double horasTrabajadas, double tarifaPorHora)
18             : horasTrabajadas(horasTrabajadas), tarifaPorHora(tarifaPorHora) {}
19         double salario() {
20             return horasTrabajadas * tarifaPorHora;
21         }
22 };
23
24 class EmpleadoFijo : public Empleado {
25     private:
26         double salarioMensual;
27     public:
28         EmpleadoFijo(double salarioMensual) : salarioMensual(salarioMensual) {}
29         double salario() {
30             return salarioMensual;
31         }
32 };
33
34
35 int main() {
36     Empleado* empleados[2];
37     empleados[0] = new EmpleadoPorHora(40, 15);
38     empleados[1] = new EmpleadoFijo(2000);
39     for (int i = 0; i < 2; i++) {
```

```

40     EmpleadoPorHora* empleadoPH = dynamic_cast<EmpleadoPorHora*>(empleados[i]);
41     if (empleadoPH) {
42         cout << "Empleado por Hora: " << empleadoPH->salario() << endl;
43     }
44     EmpleadoFijo* empleadoF = dynamic_cast<EmpleadoFijo*>(empleados[i]);
45     if (empleadoF) {
46         cout << "Empleado Fijo: " << empleadoF->salario() << endl;
47     }
48 }
49 return 0;
50 }
51

```

Listing 155. Ejemplo usando *dynamic_cast*.

En el ejemplo anterior se crea un arreglo de apuntadores a *Empleado* y se asignan las instancias de las clases derivadas *EmpleadoPorHora* y *EmpleadoFijo*. Luego, en el ciclo *for* se recorre el arreglo y se utiliza el operador *dynamic_cast* para verificar si el puntero es una instancia de *EmpleadoPorHora* o *EmpleadoFijo*, y de esta manera acceder a sus métodos específicos. El *dynamic_cast* retorna un apuntador al tipo especificado si la conversión es válida, o nulo si no es así.

18.2.5 Sobreescritura de métodos en C++

Algunas veces, los métodos heredados no cumplen completamente la función que quisiéramos que realicen en las clases derivadas. Es posible en C++ **sobreescribir** un método de la clase base en la clase derivada. Cuando se hace referencia al nombre del método, se ejecuta la versión de la clase en donde fue sobreescrita.

Es posible sin embargo, utilizar el método original de la clase base por medio del operador de resolución de alcance.

Se sugiere sobreescibir métodos que no vayan a ser empleados en la clase derivada, inclusive sin código para inhibir cualquier acción que no nos interese².

Concepto

La sobreescritura de métodos **no** es una sobrecarga porque se definen exactamente con la misma firma.

Ejemplo:

²n teoría esto no debería ser necesario anular operaciones si nos apegamos a la regla del 100% (**de conformidad con la definición**)

```
1  // EMPLOY.H
2  #ifndef EMPLOY_H
3  #define EMPLOY_H
4
5  using namespace std;
6
7  class Employee {
8      public:
9          Employee(const char*, const char*);
10         void print() const;
11         ~Employee();
12     private:
13         char *firstName;
14         char *lastName;
15 };
16 #endif /*EMPLOY_H*/
17
18 // EMPLOY.CPP
19 #include <string.h>
20 #include <iostream>
21 #include <assert.h>
22 #include "employ.h"
23
24 Employee::Employee(const char *first, const char *last){
25     firstName = new char[ strlen(first) + 1 ];
26     assert(firstName != 0);    strcpy(firstName, first);
27
28     lastName = new char[ strlen(last) + 1 ];
29     assert(lastName != 0);
30     strcpy(lastName, last);
31 }
32
33 void Employee::print() const
34 { cout << firstName << ' ' << lastName; }
35
36 Employee::~~Employee() {
37     delete [] firstName;
38     delete [] lastName;
39 }
40
41 // HOURLY.H
```

```
42 #ifndef HOURLY_H
43 #define HOURLY_H
44
45 #include "employ.h"
46
47 class HourlyWorker : public Employee {
48     public:
49         HourlyWorker(const char*, const char*, float, float);
50         float getPay() const;
51         void print() const;
52     private:
53         float wage;
54         float hours;
55 };
56
57 #endif /*HOURLY_H_*/
58
59 // HOURLY_B.CPP
60 #include <iostream>
61 #include <iomanip>
62 #include "hourly.h"
63
64 HourlyWorker::HourlyWorker(const char *first, const char *last, float initHours,
65     float initWage) : Employee(first, last) {
66     hours = initHours;
67     wage = initWage;
68 }
69
70 float HourlyWorker::getPay() const { return wage * hours; }
71
72 void HourlyWorker::print() const {
73     cout << "HourlyWorker::print()\n\n";
74
75     Employee::print();    // llama a función de clase base
76
77     cout << " es un trabajador por hora con sueldo de"
78         << " $" << setprecision(2) << getPay() << endl;
79 }
80
81 // main.CPP
82 #include <iostream>
83 #include "hourly.h"
```



```
84
85 int main() {
86     HourlyWorker h("Bob", "Smith", 40.0, 7.50);
87     h.print();
88     return 0;
89 }
```

Listing 156. Ejemplo de redefinición de métodos en C++.

18.2.6 Herencia múltiple en C++

Es posible que una clase requiere recibir miembros de más de una clase. Esto es posible haciendo uso de **herencia múltiple**.

Concepto

Herencia múltiple es la capacidad de una clase derivada de heredar miembros de varias clases base.

Sintaxis

```
class <nombre clase derivada> : <clase base 1> , <clase base 2>,
    ...<clase base n> {
    ...
};
```

Ejemplo:

```
1 class A{
2     public:
3     int i;
4     void a() {}
5 };
6
7 class B{
8     public:
9     int j;
```

```
10         void b() {}
11     };
12
13     class C{
14     public:
15         int k;
16         void c() {}
17     };
18
19     class D: public A, public B, public C {
20     public:
21         int l;
22         void d() {}
23     };
24
25     int main() {
26         D var1;
27
28         var1.a();
29         var1.b();
30         var1.c();
31         var1.d();
32         return 0;
33     }
34
```

Listing 157. Ejemplo de herencia múltiple con C++.

Ejemplo:

```
1  // BASE1.H
2  #ifndef BASE1_H
3  #define BASE1_H
4
5  class Base1 {
6  public:
7      Base1(int x) { value = x; }
8      int getData() const { return value; }
9  protected:
10     int value;
11 };
12
13 #endif /*BASE1_H*/
```

```

14
15 // BASE2.H
16 #ifndef BASE2_H
17 #define BASE2_H
18
19 class Base2 {
20     public:
21         Base2(char c) { letter = c; }
22         char getData() const { return letter; }
23     protected:
24         char letter;
25 };
26
27 #endif /*BASE2_H_*/
28
29 // DERIVED.H
30 #ifndef DERIVED_H
31 #define DERIVED_H
32
33 // herencia múltiple
34 class Derived : public Base1, public Base2 {
35     friend ostream &operator<<(ostream &, const Derived &);
36     public:
37         Derived(int, char, float);
38         float getReal() const;
39     private:
40         float real;
41 };
42
43 #endif /*DERIVED_H_*/
44
45 // DERIVED.CPP
46 #include <iostream>
47 #include "base1.h"
48 #include "base2.h"
49 #include "derived.h"
50
51 Derived::Derived(int i, char c, float f): Base1(i), Base2(c) {
52     real = f;
53 }
54
55 float Derived::getReal() const {

```

```
56         return real;
57     }
58
59     ostream &operator<<(ostream &output, const Derived &d) {
60         output << "      Entero: " << d.value
61             << "\n  Caracter: " << d.letter
62             << "\nNúmero real: " << d.real;
63
64         return output;
65     }
66
67     // main.CPP
68     #include <iostream>
69     #include "base1.h"
70     #include "base2.h"
71     #include "derived.h"
72
73     int main() {
74         Base1 b1(10), *base1Ptr;
75         Base2 b2('Z'), *base2Ptr;
76         Derived d(7, 'A', 3.5);
77
78         cout << "Objeto b1 contiene entero "
79             << b1.getData()
80             << "\nObjeto b2 contiene caracter "
81             << b2.getData()
82             << "\nObjeto d contiene:\n" << d;
83         cout << "\n\nmiembros de clase derivada pueden ser"
84             << " accesados individualmente:"
85             << "\n  Entero: " << d.Base1::getData()
86             << "\n  Caracter: " << d.Base2::getData()
87             << "\n  Número real: " << d.getReal() << "\n\n";
88         // Probar: cout<<d.getData(); Es un error?
89         cout << "Miembros derivados pueden ser tratados como "
90             << "objetos de su clase base:\n";
91
92         base1Ptr = &d;
93         cout << "base1Ptr->getData() "
94             << base1Ptr->getData();
95
96         base2Ptr = &d;
97         cout << "\nbase2Ptr->getData() "
```

```
98         << base2Ptr->getData() << endl;
99         return 0;
100     }
```

Listing 158. Otro ejemplo de herencia múltiple en C++.

Aquí se tiene un problema de **ambigüedad** al heredar dos métodos con el mismo nombre de clases diferentes. Se resuelve poniendo antes del nombre del miembro el nombre de la clase: *objeto*. < *clase* ::> *miembro*. El nombre del objeto es necesario, pues no se está haciendo referencia a un miembro estático.

Ambigüedades

En el ejemplo anterior se vio un caso de ambigüedad al heredar de clases distintas un miembro con el mismo nombre. Normalmente se deben tratar de evitar esos casos, pues vuelven confusa nuestra jerarquía de herencia.

Existen otros casos donde es posible que se de la ambigüedad.

Ejemplo:

```
1  //ejemplo de ambigüedad en la herencia
2  class B{
3  public:
4      int b;
5  };
6
7  class D: public B, public B { //error
8  };
9
10 void f( D *p) {
11     p->b=0; //ambiguo
12 }
13
14 int main() {
15     D obj;
16     f(&obj);
17     return 0;
18 }
```

Listing 159. Ejemplo de ambigüedad en herencia múltiple con C++.

El código anterior tiene un error en la definición de herencia múltiple, ya que no es posible heredar más de una vez una misma clase de manera directa. Sin embargo, si es posible heredar las características de una clase más de una vez indirectamente:

Ejemplo:

```
1  //ejemplo de ambigüedad en la herencia
2  #include <iostream>
3  using namespace std;
4
5  class A{
6      public:
7      int next;
8  };
9
10 class B: public A{
11 };
12
13 class C: public A{
14 };
15
16 class D: public B, public C {
17     int g();
18 };
19
20 int D::g() {
21     //next=0; Error: asignación ambigua
22     return B::next == C::next;
23 }
24
25 class E: public D{
26     public:
27     int x;
28     int getx() {
29         return x;
30     }
31 };
32
33 int main() {
34     D obj;
35     E obje;
36
37     obj.B::next=10;
38     obj.C::next=20;
39     // obj.A::next=11; Error: acceso ambiguo
40     // obj.next=22; Error: acceso ambiguo
```

```

41     cout<<"next de B: "<<obj.B::next<<endl;
42     cout<<"next de C: "<<obj.C::next<<endl;
43
44     obje.x=0;
45     obje.B::next=11;
46     obje.C::next=22;
47     cout<<"obje next de B: "<<obje.B::next<<endl;
48     cout<<"obje next de C: "<<obje.C::next<<endl;
49     return 0;
50 }

```

Listing 160. Ejemplo 2 de ambigüedad en herencia múltiple con C++.

Este programa hace que las instancias de la clase D tengan objetos de clase base duplicados y provoca los accesos ambiguos. Este problema se resuelve con **herencia virtual**.

Concepto

Herencia de clase base virtual: Si se especifica a una clase base como virtual, solamente un objeto de la clase base existirá en la clase derivada.

Para el ejemplo anterior, las clases B y C deben declarar a la clase A como clase base virtual:

Sintaxis

```

class B: virtual public A {...}

class C: virtual public A {...}

```

El acceso entonces a los miembros puede hacerse usando una de las clases de las cuales heredo el miembro:

```

1 obj.B::next=10;
2 obj.C::next=20;

```

O simplemente accediéndolo como un miembro no ambiguo:ejercicio: probar los ejemplo y modificar la definición a clases base virtuales.

```
obj.next=22;
```

En cualquier caso se tiene solo una copia del miembro, por lo que cualquier modificación del atributo *next* es sobre una única copia del mismo.

Constructores en herencia múltiple

Si hay constructores con argumentos, es posible que sea necesario llamarlos desde el constructor de la clase derivada. Para ejecutar los constructores de las clases base, pasando los argumentos, es necesario especificarlos después de la declaración de la función de construcción de la clase derivada, separados por coma.

Sintaxis

```
<Constructor clase derivada>(<argumentos>) : <base1> (<argumentos>),  
    <base2> (<argumentos>), ... , <basen> (<argumentos>) {  
    ...  
}
```

Donde como en la herencia simple, el nombre base corresponde al nombre de la clase, o en este caso, clases base.

El orden de llamada a constructores de las clases base se puede alterar a conveniencia. Una **excepción** a considerar es cuando se resuelve ambigüedad de una clase base pues en ese caso el constructor de la clase base ambigua únicamente se ejecuta una vez. Si no es especificado por el programador, se ejecuta el constructor sin parámetros de la clase base ambigua. Si se requiere pasar parámetros, se debe especificar la llamada antes de las llamadas a constructores de clase base directas. Este es el único caso en que es posible llamar a un constructor de una clase que no es un ancestro directo[?].

18.3 Herencia: implementación en Java

La clase de la cual se toman sus características se conoce como **superclase**; mientras que la clase que ha sido creada a partir de la clase base se conoce como **subclase**.

La herencia en Java difiere ligeramente de la sintaxis de implementación de herencia en C++.

Sintaxis

```
class <subclase> extends <superclase> {  
    //cuerpo subclase  
}
```

Ejemplo:

Una clase vehículo que describe a todos aquellos objetos vehículos que viajan en carreteras. Puede describirse a partir del número de ruedas y de pasajeros. De la definición de vehículos podemos definir objetos más específicos (especializados). Por ejemplo la clase camión.

```
1 //ejemplo de herencia  
2 class Vehiculo{  
3     private int ruedas;  
4     private int pasajeros;  
5  
6     public void setRuedas (int num) {  
7         ruedas=num;  
8     }  
9  
10    public int getRuedas () {  
11        return ruedas;  
12    }  
13  
14    public void setPasajeros (int num) {  
15        pasajeros=num;  
16    }  
17  
18    public int getPasajeros () {
```

```
19         return pasajeros;
20     }
21
22 }
23
24 //clase Camion con herencia de Vehiculo
25 public class Camion extends Vehiculo {
26     private int carga;
27
28     public void setCarga(int num) {
29         carga=num;
30     }
31
32     public int getCarga() {
33         return carga;
34     }
35
36     public void muestra() {
37         // uso de métodos heredados
38         System.out.println("Ruedas: " + getRuedas());
39         System.out.println("Pasajeros: " + getPasajeros());
40         // método de la clase Camion
41         System.out.println("Capacidad de carga: " + getCarga());
42     }
43
44     public static void main(String argsv[]) {
45         Camion ford= new Camion();
46         ford.setRuedas(6);
47         ford.setPasajeros(3);
48         ford.setCarga(3200);
49         ford.muestra();
50     }
51
52 }
```

Listing 161. Ejemplo de herencia en Java.

En el programa anterior se puede apreciar claramente como una clase Vehículo hereda sus características a la subclase *Camion*, pudiendo este último aprovechar recursos que no declara en su definición.

Tema sugerido

BlueJ en apéndice A. (??)

18.3.1 Clase *Object*

En Java toda clase que se define tiene herencia implícita de una clase llamada *Object*. En caso de que la clase que crea el programador defina una herencia explícita a una clase, hereda las características de la clase *Object* de manera indirecta³.

Ver clase *Object* en: <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

18.3.2 Control de acceso a miembros en Java

Existen tres palabras reservadas para el control de acceso a los miembros de una clase: *public*, *private* y *protected*. Estas sirven para proteger los miembros de la clase en diferentes formas.

El control de acceso, como ya se vio anteriormente, se aplica a los métodos, atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

- *private*. Un miembro privado únicamente puede ser utilizado por los métodos miembro de la clase donde fue declarado. Un miembro privado no es posible que sea manejado ni siquiera en sus subclases.
- *protected*. Un miembro protegido puede ser utilizado únicamente por los métodos miembro de la clase donde fue declarado, por los métodos miembro de las clases derivadas o clases que pertenecen al mismo paquete. El acceso protegido es como un nivel intermedio entre el acceso privado y público.
- *public*. Un miembro público puede ser utilizado por cualquier método. Este es visible en cualquier lugar que la clase sea visible

Ejemplo:

```
1 //ejemplo de control de acceso
2
3 class Acceso{
4     protected int b;
5     protected int f2 () {
```

³En este caso hay que considerar que las características de la clase *Object* pudieron haber sido modificadas a través de la jerarquía de herencia.

```
6         return b;
7     }
8
9     private int c;
10    private int f3() {
11        return c;
12    }
13
14    public int d, f;
15    public int f4() {
16        return d;
17    }
18
19 }
20
21 public class EjemploAcceso {
22
23     public static void main(String argsv[]) {
24         Acceso obj= new Acceso();
25
26         obj.f2(); //es válido, ya que por omisión
27         obj.b=2; //las dos clases están en el mismo paquete
28
29         obj.c=3; //error es un atributo privado
30         obj.f3(); //error es un método privado
31
32         obj.d=5;
33         obj.f4();
34     }
35 }
36
```

Listing 162. Ejemplo de control de acceso en Java.

El ejemplo anterior genera errores de compilación al tratar de acceder desde otra clase a miembros privados. Sin embargo, los miembros protegidos si pueden ser accedidos porque están considerados implícitamente dentro del mismo paquete.

18.3.3 Control de acceso de clase public en Java

Este controlador de acceso *public*, opera a nivel de la clase para que esta se vuelva visible y accesible desde cualquier lugar, lo que permitiría a cualquier otra

clase hacer uso de los miembros de la clase pública.

Sintaxis
<pre>public class TodosMeVen { // definición de la clase }</pre>

La omisión de este calificador limita el acceso a la clase para que solo sea utilizada por clases pertenecientes al mismo paquete.

Además, se ha mencionado que en un archivo fuente únicamente puede existir una clase pública, la cual debe coincidir con el nombre del archivo. La intención es que cada clase que tenga un objetivo importante debería ir en un archivo independiente, pudiendo contener otras clases no públicas que le ayuden a llevar a cabo su tarea.

18.3.4 Constructores de superclase

Los constructores no se heredan a las subclases. El constructor de la superclase puede ser llamado desde la clase derivada, para inicializar los atributos heredados y no tener que volver a introducir código de inicialización ya escrito en la superclase.

La llamada explícita al constructor de la superclase se realiza mediante la referencia **super** seguida de los argumentos –si los hubiera– del constructor de la clase base. La llamada a este constructor debe ser hecha en la primera línea del constructor de la subclase. Si no se introduce así, el constructor de la clase derivada llamará automáticamente al constructor por omisión (sin parámetros) de la superclase.

En el ejemplo siguiente podrá apreciarse esta llamada al constructor de la superclase.

18.3.5 Manejo de objetos de subclase como objetos de superclase en Java

Un objeto de una clase derivada, puede ser manejado como un objeto de su superclase. Sin embargo, un objeto de la clase base no es posible tratarlo como un objeto de clase derivada.

Un objeto de una subclase puede ser asignado a una variable de referencia de su superclase sin necesidad de indicar una conversión explícita mediante enmascaramiento. Cuando si se necesita utilizar enmascaramiento es para asignar de vuelta un objeto que aunque sea de una clase derivada, este referenciado por una variable de clase base. Esta conversión explícita es verificada por la máquina virtual, y si no corresponde el tipo real del objeto, no se podrá hacer la asignación y se generará una excepción en tiempo de ejecución.

Ejemplo:

```
1 Superclase s= new superclase(), aptSuper;
2 Subclase sub= new subclase(), aptSub;
3
4 //válido
5 aptSuper = sub;
6
7 aptSub = (Subclase) aptSuper;
8
9 //inválido
10 aptSub= (Subclase) s;
```

Podemos ver en el ejemplo anterior, que un objeto puede “navegar” en la jerarquía de clases hacia sus superclases, pero no puede ir a una de sus subclases, ni utilizando el enmascaramiento. Esto se hace por seguridad, ya que la subclase seguramente contendrá un mayor número de elementos que una instancia de superclase y estos no podrían ser utilizados porque causarían una inconsistencia.

Por último, es importante señalar que mientras un objeto de clase derivada este referenciado como un objeto de superclase, deberá ser tratado como si el objeto fuera únicamente de la superclase; por lo que no podrá en ese momento tener referencias a atributos o métodos definidos en la clase derivada.

Un código completo se muestra a continuación. [Ejemplo:](#)

```
1 // definición de clase point
2 public class Point {
3     protected double x, y; // coordenadas del punto
4
5     // constructor
6     public Point( double a, double b ) {
7         setPoint( a, b );
8     }
9
10    // asigna a x,y las coordenadas del punto
```

```
11     public void setPoint( double a, double b ) {
12         x = a;
13         y = b;
14     }
15
16     // obtiene coordenada x
17     public double getX() {
18         return x;
19     }
20
21     // obtiene coordenada y
22     public double getY() {
23         return y;
24     }
25
26     // convierte información a cadena
27     public String toString(){
28         return "[" + x + ", " + y + "]";
29     }
30 }
31 // Definición de clase círculo
32 public class Circle extends Point { // Hereda de Point
33     protected double radius;
34
35     // constructor sin argumentos
36     public Circle() {
37         super( 0, 0 ); // llamada a constructor de clase base
38         setRadius( 0 );
39     }
40
41     // Constructor
42     public Circle( double r, double a, double b ) {
43         super( a, b ); // llamada a constructor de clase base
44         setRadius( r );
45     }
46
47     // Asigna radio del círculo
48     public void setRadius( double r )
49     { radius = ( r >= 0.0 ? r : 0.0 ); }
50
51     // Obtiene radio del círculo
52     public double getRadius() { return radius; }
```

```
53
54 // Cálculo área del círculo
55 public double area() { return 3.14159 * radius * radius; }
56
57 // Convierte información en cadena
58 public String toString() {
59     return "Centro = " + "[" + x + ", " + y + "]" +
60         "; Radio = " + radius;
61 }
62 }
63 // Clase de Prueba de las clases Point y Circle
64 public class Prueba {
65
66     public static void main( String args[] ) {
67         Point pointRef, p;
68         Circle circleRef, c;
69
70         p = new Point( 3.5, 5.3 );
71         c = new Circle( 2.7, 1.2, 8.9 );
72
73         System.out.println( "Punto p: " + p.toString() );
74         System.out.println( "Círculo c: " + c.toString() );
75
76         // Tratamiento del círculo como instancia de punto
77         pointRef = c; // asigna círculo c a pointRef
78         // en realidad Java lo reconoce dinámicamente como objeto Circle
79         System.out.println( "Círculo c (via pointRef): " + pointRef.toString() );
80
81         // Manejar a un círculo como círculo
82         // (obteniéndolo de una referencia de punto)
83         // asigna círculo c a pointref. Se repite la operación por claridad
84         pointRef = c;
85         circleRef = (Circle) pointRef; // enmascaramiento de superclase a subclase
86         System.out.println( "Círculo c (via circleRef): " + circleRef.toString() );
87         System.out.println( "Área de c (via circleRef): " + circleRef.area() );
88
89         // intento de referenciar a un objeto point
90         // desde una referencia de Circle (genera una excepción)
91         circleRef = (Circle) p;
92     }
93 }
```


Listing 163. Ejemplo de objetos de sub clase como de superclase en Java.

18.3.6 Sobreescritura de métodos (Overriding)

Frecuentemente, los métodos heredados no implementan el comportamiento específico requerido por la subclase. Java permite **sobreescibir** (o sobreescribir) un método de la clase base en la clase derivada. Al invocar el método desde una instancia de la subclase, se ejecutará la nueva versión.

Es una buena práctica utilizar la anotación `@Override` para asegurar que el compilador verifique la firma del método. Sin embargo, aún es posible acceder a la lógica original de la clase base utilizando la referencia `super`.

Ejemplo:

```
1  // Clase base: Empleado
2  public class Empleado {
3      // Se recomienda usar protected si se planea acceso directo en herencia,
4      // o mantener private y usar getters.
5      protected String nombre;
6      protected String apellido;
7
8      public Empleado(String nombre, String apellido){
9          // Asignación directa, no es necesario new String()
10         this.nombre = nombre;
11         this.apellido = apellido;
12     }
13
14     public void imprimirDatos() {
15         System.out.println("Empleado: " + nombre + " " + apellido);
16     }
17 }
18
19 // Clase derivada: TrabajadorPorHora
20 public class TrabajadorPorHora extends Empleado {
21     private double sueldoHora; // double tiene mayor precisión que float
22     private double horas;
23
24     public TrabajadorPorHora(String nombre, String apellido,
25                             double horas, double sueldoHora) {
26         super(nombre, apellido); // Llama al constructor de la clase base
27         this.horas = horas;
28         this.sueldoHora = sueldoHora;
29     }
30
31     public double calcularPago() {
32         return sueldoHora * horas;
```

```

33     }
34
35     @Override // Buena práctica: asegura que estamos sobrescribiendo
36     public void imprimirDatos() {
37         System.out.println("--- Detalle Trabajador por Hora ---");
38
39         // Reutilizamos la lógica de la clase padre
40         super.imprimirDatos();
41
42         System.out.println("Sueldo calculado: $" + calcularPago());
43     }
44 }
45
46 // Clase de prueba
47 public class TestEmpleados {
48
49     public static void main(String[] args) {
50         Empleado e = new Empleado("Juan", "García");
51         TrabajadorPorHora h = new TrabajadorPorHora("Ana", "Pérez", 40.0, 15.50);
52
53         e.imprimirDatos(); // Ejecuta versión de Empleado
54         h.imprimirDatos(); // Ejecuta versión sobrescrita de TrabajadorPorHora
55     }
56 }

```

Listing 164. Ejemplo de sobrescritura de métodos y uso de `super`.

18.3.7 Calificador *final*

Es posible que tengamos la necesidad de que cierta parte de una clase no pueda ser modificada en futuras extensiones de la jerarquía de herencia. Para esto es posible utilizar el calificador *final*.

Si un método se especifica en una clase X como *final*:

Sintaxis
<acceso> final <tipo> nombreMétodo(<parámetros>)

Se está diciendo que el método no podrá ser sobrescrito en las subclases de X.

Aunque se omita este calificador, si se trata de un método de clase (estático) o privado, se considera *final* y no podrá ser sobrescrito.

Por otro lado, es posible que no queramos dejar la posibilidad de extender una clase, para lo que se utiliza el calificador *final* a nivel de clase:

Sintaxis

```
<acceso> final class nombreClase {  
  
    //definición de la clase  
  
}
```

De esta forma, la clase no permite generar subclases a partir de ella. De hecho, el API de Java incluye muchas clases *final*, por ejemplo la clase `java.lang.String` no puede ser especializada.

18.3.8 Interfaces en Java

Java únicamente cuenta con manejo de herencia simple, y la razón que se ofrece es que la herencia múltiple presenta algunos problemas de ambigüedad que complica el entendimiento del programa, sin que este tipo de herencia justifique las ventajas obtenidas de su uso.

Sin embargo, es posible que se necesiten recibir características de más de un origen. Java soluciona esto mediante el uso de interfaces, que son una forma para declarar tipos especiales de clase que, aunque con ciertas limitaciones, no ofrecen las complicaciones de la herencia múltiple.

Una interfaz tiene un formato muy similar a una clase, sus principales características:

- Una interfaz proporciona los nombres de los métodos (método abstracto), pero no sus implementaciones⁴.
- Los métodos son públicos por omisión.
- Una clase puede implementar varias interfaces, aunque solo pueda heredar una clase.
- No es posible crear instancias de una interfaz.
- La clase que implementa la interfaz debe escribir el código de todos los métodos, de otra forma no se podrá generar instancias de esa clase.

El formato general para la declaración de una interfaz es el siguiente:

⁴En esta caso si se considera la declaración de prototipos.

Sintaxis

```
[public] interface <nombreInterfaz> {  
    //descripción de miembros  
    //los métodos no incluyen código:  
    <acceso> <tipo> <nombreMetodo> ( <parámetros> ) ;  
}
```

El cuerpo de la interfaz generalmente es una lista de prototipos de métodos, pero puede contener atributos si se requiere⁵.

Una clase implementa una interfaz a través de la palabra reservada *implements* después de la especificación de la herencia (si la hubiera) :

Sintaxis

```
class <SubClase> extends <Superclase> implements <nombreInterfaz> {  
    //definición de la clase  
    //debe incluirse la definición de los métodos de la interfaz  
    //con la implementación del código de dichos métodos.  
}
```

Además, una interfaz puede ser extendida de la misma forma que una clase, aprovechando las interfaces previamente definidas, mediante el uso de la cláusula *extends*.

⁵El parámetro debe incluir el nombre, el cual no es obligatorio que coincida en la implementación.

Sintaxis

```
[public] interface <nombreInterfaz> extends <InterfazBase> {  
  
    //descripción de miembros  
  
}
```

De forma distinta a la jerarquía de clases, donde se tiene una jerarquía lineal que parte siempre de una clase simple *Object*, una clase soporta herencia múltiple de interfaces, resultando en una jerarquía con múltiples raíces de diferentes interfaces.

Ejemplo:

```
1  //interfaz  
2  interface IStack {  
3      void push(Object item);  
4      Object pop();  
5  }  
6  
7  //clase implementa la interfaz  
8  class StackImpl implements IStack {  
9      protected Object[] stackArray;  
10     protected int tos;  
11  
12     public StackImpl(int capacity) {  
13         stackArray = new Object[capacity];  
14         tos = -1;  
15     }  
16  
17     //implementa el método definido en la interfaz  
18     public void push(Object item)  
19     { stackArray[++tos] = item; }  
20  
21     //implementa el método definido en la interfaz  
22     public Object pop() {  
23         Object objRef = stackArray[tos];  
24         stackArray[tos] = null;  
25         tos--;  
26         return objRef;  
}
```

```
27     }
28
29     public Object peek() { return stackArray[tos]; }
30 }
31
32 // extendiendo una interfaz
33 interface ISafeStack extends IStack {
34     boolean isEmpty();
35     boolean isFull();
36 }
37
38
39 //esta clase hereda la implementación de la pila StackImpl
40 // e implementa la nueva interfaz extendida ISafeStack
41 class SafeStackImpl extends StackImpl implements ISafeStack {
42
43     public SafeStackImpl(int capacity) { super(capacity); }
44
45     //implementa los métodos de la interfaz
46     public boolean isEmpty() { return tos < 0; }
47     public boolean isFull() { return tos >= stackArray.length; }
48 }
49
50
51 public class StackUser {
52
53     public static void main(String args[]) {
54         SafeStackImpl safeStackRef = new SafeStackImpl(10);
55         StackImpl stackRef = safeStackRef;
56         ISafeStack isafeStackRef = safeStackRef;
57         IStack istackRef = safeStackRef;
58         Object objRef = safeStackRef;
59
60         safeStackRef.push("Dolar");
61         stackRef.push("Peso");
62
63         // tipo de dato simple es convertido a Integer:
64         stackRef.push(1);
65         System.out.println(stackRef.peek().getClass());
66
67         System.out.println(isafeStackRef.pop());
68         System.out.println(istackRef.pop());
```

```
69
70     System.out.println(objRef.getClass());
71 }
72 }
```

Listing 165. Ejemplo de interfaz en Java.

Por otro lado, una interfaz también puede ser utilizada para definir nuevos tipos. Una interfaz así o una clase que implementa a una interfaz de este estilo es conocida como **Supertipo**.

Es importante resaltar tres diferencias en las relaciones de herencia y como esta funciona entre clases e interfaces:

1. **Implementación lineal de jerarquía de herencia entre clases:** una clase extiende a otra clase.
2. **Jerarquía de herencia múltiple entre interfaces:** una interfaz extiende otras interfaces.
3. **Jerarquía de herencia múltiple entre interfaces y clases:** una clase implementa interfaces.

Interfaces en Java con implementación predeterminada de métodos

A partir de Java 8 permite el uso de métodos implementados en interfaces. Estos pueden ser métodos estáticos o un tipo especial de método llamado *Default*⁶.

Sintaxis

```
[public] interface <nombreInterfaz> {
    //implementación predeterminada de método
    <acceso> default <tipo> <nombreMetodo> ( <parámetros> ) {
        //código del método
    }
}
```

Ahora, las clases que implementan la interfaz no tienen que proporcionar su propia implementación del método o métodos. Si no lo hacen, se utilizará la implementación predeterminada.

⁶<https://www.techempower.com/blog/2013/03/26/everything-about-java-8/>, <https://stackoverflow.com/questions/18286235/what-is-the-default-implementation-of-method-defined-in-an-interface>

Ambigüedad con implementación de métodos en interfaces

Si una clase hereda de una interfaz que tiene un método por defecto y también hereda de una clase base que tiene un método con la misma firma, se utilizará la implementación de la clase base, y no se generará una ambigüedad.

Pero si de dos interfaces distintas se hereda un método implementado con la misma firma, si existe ambigüedad. Para evitarla, la clase que implementa a las interfaces tiene que proporcionar su propia implementación del método:

Ejemplo: [Ejemplo](#):

```
1
2 interface InterfazA {
3     default void metodo() {
4         System.out.println("InterfazA");
5     }
6 }
7
8 interface InterfazB {
9     default void metodo() {
10        System.out.println("InterfazB");
11    }
12 }
13
14 // Ambigüedad si descomentas esta línea:
15 // class MiClase implements InterfazA, InterfazB {}
16
17 class MiClase implements InterfazA, InterfazB {
18     // Si proporcionas tu propia implementación, no hay ambigüedad
19     @Override
20     public void metodo() {
21         System.out.println("Implementación en MiClase");
22     }
23 }
24
25 public class Main {
26     public static void main(String[] args) {
27         MiClase instancia = new MiClase();
28         instancia.metodo(); // Imprime "Implementación en MiClase"
29     }
30 }
31
```


Listing 166. Ejemplo de interfaz con potencial ambigüedad en métodos psobreescritos en Java.

Y ¿atributos en las interfaces?

Hasta Java 7, las interfaces solo podían contener constantes (variables estáticas finales) como atributos. Estos atributos se consideraban **automáticamente** como *public*, *static*, y *final*. Un ejemplo sería:

```
1 public interface MiInterfaz {  
2     int MI_CONSTANTE = 42; // Atributo (public, static, final)  
3 }
```

18.4 Herencia: Implementación en Ruby

Ruby, al igual que Java, no cuenta con herencia múltiple, por lo que únicamente es posible indicar una clase como superclase. Para esto se utiliza el símbolo < antes del nombre de la superclase.

Sintaxis

```
class <SubClase> < <SuperClase>
  # codigo de la subclase
end
```

Retomando el ejemplo de la jerarquía de Vehículo y Camión, visto anteriormente.

```
1  # ejemplo de herencia
2  class Vehiculo
3      @ruedas
4      @pasajeros
5      attr_accessor :ruedas, :pasajeros
6  end
7
8  # clase Camion con herencia de Vehiculo
9  class Camion < Vehiculo
10     @carga
11     attr_accessor :carga
12
13     public
14     def muestra
15         # uso de métodos heredados y generados automáticamente
16         # por attr_accessor
17
18         puts "Ruedas: #{ruedas}"
19         puts "Pasajeros: #{pasajeros}"
20         # método de la clase Camion
21         puts "Capacidad de carga: #{carga}"
22     end
23 end
```

```
24
25 # Código de prueba
26 ford= Camion.new
27 ford.ruedas =6
28 ford.pasajeros=3
29 ford.carga=3200
30 ford.muestra
```

Listing 167. Ejemplo de herencia en Ruby.

18.4.1 Clase *Class*

Las clases en Ruby son considerados objetos de primera clase⁷. Cada clase declarada en Ruby es una instancia de la clase *Class* [?].

El proceso que se sigue es el siguiente: cuando una clase es definida, un objeto de la clase *Class* es creado y asignado a una constante global con el nombre usado en la definición. Cuando un nuevo objeto de la clase es instanciado (*nombreClase.new*), el método *new* de *Class* es ejecutado por omisión.⁸

A continuación se presenta la información general de la clase *Class* [?]:

```
class methods
  inherited aClass.inherited( aSubClass )
  This is a singleton method (per class) invoked by Ruby when a subclass
  of aClass is created. The new subclass is passed as a parameter.
```

```
class Top
  def Top.inherited(sub)
    print "New subclass: ", sub, "
    n"
  end
end
```

```
class Middle < Top
end
```

```
class Bottom < Middle
end
```

⁷Pueden ser usados en el lenguaje de programación como cualquier otro objeto.

⁸De hecho, el método *new* de la clase *Class* puede ser sobrescrito, aunque no es una práctica recomendada.

produces:

```
New subclass: Middle
New subclass: Bottom
```

```
new Class.new( aSuperClass=Object ) -> aClass
Creates a new anonymous (unnamed) class with the given superclass (or
Object if no parameter is given).
```

```
instance methods
new aClass.new( [ args ]* ) -> anObject
Creates a new object of aClass's class, then invokes that object's
initialize method, passing it args.
```

```
superclass aClass.superclass -> aSuperClass or nil
Returns the superclass of aClass, or nil.
```

```
Class.superclass » Module
Object.superclass » nil
```

18.4.2 Clase *Object*

La clase padre de todas las clases en Ruby es la clase *Object*, por lo que sus métodos están disponibles para todos los objetos a menos que estos se encuentren sobrescritos.

A continuación se presenta una lista de los métodos de *Object*:

```
==    ===    =~    __id__    __send__    class    clone    dclone    display
dup    enum_for    eql?    equal?    extend    freeze    frozen?    hash
id    inspect    instance_eval    instance_of?    instance_variable_defined?
instance_variable_get    instance_variable_get    instance_variable_set
instance_variable_set    instance_variables    is_a?    kind_of?    method
methods    new    nil?    object_id    private_methods    protected_methods
public_methods    remove_instance_variable    respond_to?    send
singleton_method_added    singleton_method_removed    singleton_method_unde
singleton_methods    taint    tainted?    to_a    to_enum    to_s    to_yaml
to_yaml_properties    to_yaml_style    type    untaint
```

18.4.3 Control de acceso a miembros en Ruby

Como ya se ha mencionado, también en Ruby existen tres palabras reservadas para el control de acceso a los **métodos** de una clase: *public*, *private* y *protected*. Estas

sirven para proteger los métodos de la clase en diferentes formas.

Resumen de tipos de acceso:

- **private**. Un método **privado** no puede ser llamado mediante un objeto. Por esta razón, los métodos privados solo pueden ser utilizados en la definición de la clase.
- **protected**. Un método **protegido** puede ser utilizado únicamente por objetos de la clase donde se define el método, así como por sus subclases.
- **public**. Un método **público** puede ser utilizado por cualquier método. Éste es visible en cualquier lugar que la clase sea visible. Los métodos son públicos por omisión, con excepción del método `initialize`, el cual es privado.

El comportamiento de acceso privado y protegido es diferente de los lenguajes C++ y Java. En Ruby, un método protegido puede ser llamado por cualquier instancia de la clase donde se define o de sus subclases. Por su parte, un método privado puede ser llamado solamente en el contexto un objeto de la clase. Los métodos privados no puede ser ejecutados directamente ni siquiera por otro objeto de la misma clase [?]. El uso de *self* tampoco es válido para métodos privados.

Ejemplo:

```
1 class MiClase
2     def metodo1      # default es 'public'
3         #...
4     end
5
6     protected        # subsecuentes métodos serán 'protected'
7
8     def metodo2      # 'protected'
9         #...
10    end
11
12    private           # subsecuentes métodos serán 'private'
13
14    def metodo3      # 'private'
15        #...
16    end
17
18    public            # subsecuentes métodos serán 'public'
19
20    def metodo4      # 'public'
21        #...
```

```
22     end
23 end
```

```
    O
```

```
1  class MiClase
2    def metodo1
3    end
4
5    ...
6    public      :metodo1, :metodo4
7    protected  :metodo2
8    private    :metodo3
9  end
```

18.4.4 Inicializadores de superclase

En Ruby, de manera similar a Java, es posible llamar al método inicializador de la superclase usando *super* seguido de la lista de argumentos.

```
1  class Persona
2
3    attr_accessor :nombre, :edad, :sexo
4
5    def initialize(nombre, edad, sexo)
6      @nombre, @edad, @sexo = nombre, edad, sexo
7    end
8
9  end
10
11
12 class Estudiante < Persona
13
14   attr_accessor :matr, :horas
15
16   def initialize(nombre, edad, sexo, matr, horas)
17     super(nombre, edad, sexo)
18     @matr = matr
19     @horas = horas
20   end
21
22   # ...
```

```
23
24 end
25
26 # Creando dos objetos
27 a = Persona.new("Ironman", 37, "m")
28 b = Estudiante.new("Spiderman", 36, "m", "000-13-5031", 24)
29
```

Listing 168. Ejemplo de herencia e inicializador en Ruby.

18.4.5 Manejo de objetos de la subclase como objetos de una superclase en Ruby

El manejo de objetos de subclase como objetos de superclase no es posible en Ruby, debido a que los identificadores no tienen un tipo de dato.

```
1 # definicion de clase point
2 class Point
3   attr_reader :x, :y
4
5   protected
6   @x
7   @y # coordenadas del punto
8   # constructor
9   def initialize a, b
10     setPoint a, b
11   end
12   # asigna a x,y las coordenadas del punto
13   public
14   def setPoint a, b
15     @x = a
16     @y = b
17   end
18   # convierte informacion a cadena
19   def to_s
20     return "[ #{@x}, #{@y} ]"
21   end
22 end
23
24 # Definicion de clase circulo
25 class Circle < Point # Hereda de Point
26   attr_reader :radius
```

```
27 @radius
28
29 # Constructor
30 def initialize r, a, b
31     super a, b # llamada a constructor de clase base
32     setRadius r
33 end
34
35 # Asigna radio del circulo
36 def setRadius r
37     @radius = ( r >= 0 ? r : 0 )
38 end
39
40 # Calculo area del circulo
41 def area
42     return 3.14159 * radius * radius
43 end
44
45 # Convierte informacion en cadena
46 def to_s
47     return "Centro = [ #{@x}, #{@y} ]; Radio = #{@radius}"
48 end
49 end
50
51 # código de Prueba de las clases Point y Circle
52 def prueba
53     p = Point.new 3.5, 5.3
54     c = Circle.new 2.7, 1.2, 8.9
55     puts "Punto p: " + p.to_s
56     puts "Circulo c: " + c.to_s
57     # Tratamiento del circulo como instancia de punto
58     pointRef = c; # asignar circulo c a pointRef no hace diferencia porque
59     # los identificadores no tienen tipo
60     # en realidad Ruby lo reconoce dinámicamente como objeto Circle sin importar
61     # el identificador
62     puts "Circulo c (via pointRef): " + pointRef.to_s
63
64     # asigna circulo c a pointref. Se repite la operacion por claridad
65     pointRef = c;
66     circleRef = pointRef # es asignación de referencias sin importar el tipo
67 end
68
```



```
69 prueba
```

Listing 169. Ejemplo de objetos de sub clase como de superclase en Ruby.

18.4.6 Sobreescritura de métodos

Ruby permite **sobreescribir** un método de la clase base en la clase derivada. Cuando se hace referencia al nombre del método, se ejecuta la versión de la clase en donde fue sobreescrita. Es posible sin embargo, utilizar el método de la clase base por medio la palabra *super*. A diferencia de Java, aquí solo es necesaria esta palabra (y si se necesita la lista de argumentos) para llamar al método correspondiente de la clase base desde la clase derivada.

```
1  #clase empleado
2  class Empleado
3      @firstName
4      @lastName
5
6      def initialize first, last
7          @firstName = String.new(first)
8          @lastName = String.new(last)
9      end
10
11     def print
12         puts @firstName.to_s + " " + @lastName.to_s
13     end
14 end
15
16 #clase TrabajadorporHora
17 class TrabajadorPorHora < Empleado
18     @wage
19     @hours
20
21     def initialize first, last, initHours, initWage
22         super first, last
23         @hours = initHours
24         @wage = initWage
25     end
26
27     def getPay
28         @wage * @hours
```

```
29         end
30
31     def print
32         puts "Metodo print de Trabajador por hora"
33         super # llama a función de clase base
34         puts " es un trabajador por hora con sueldo de" + " $" + getPay.to_s
35     end
36 end
37
38 #prueba de clases
39 e= Empleado.new "nombre", "apellido"
40
41 h= TrabajadorPorHora.new "Juanito", "Perez", 40.0, 7.50
42 e.print
43 h.print();
```

Listing 170. Ejemplo de redefinición de métodos en Ruby.

18.4.7 Mixins en Ruby

En lugar de interfaces, Ruby tiene *mixins*, que es la capacidad de un módulo de ser añadido a una clase o a otro módulo con la operación *include* [?]. De hecho, un *mixin* en Ruby puede verse como una interfaz pero con sus métodos implementados. Además, es posible heredar características de uno o más *mixins* (módulos) obteniendo con esto un resultado similar al de la herencia múltiple. El concepto de *mixin* apareció en el lenguaje *Flavors*. El nombre fue inspirado en una máquina de helados que ofrecía en sabores básicos mezclados con ingredientes extras, estos helados eran llamados “*Mix-In*”.⁹

Aunque también pueden ser añadidos a un objeto específico usando *extend*.

Es importante señalar que en la definición del módulo pueden incluirse atributos y restricciones de acceso, mismos que serán pasados a la clase que incluye dicho módulo. Una vez definido el módulo, éste puede ser añadido a una clase de la siguiente forma:

⁹“For a real ice cream treat, try Steve’s Ice Cream (191 Elm Street, Sommerville). Steve’s homemade ice cream is perhaps the best in town. Cones are \$.35 and \$.55, and for \$.10 each you can choose “mix-ins,” consisting of M&M’s, Heath bars, and many others. If you want fruit mixed in it costs \$.10 more. Sundaes are \$.75 and \$1.25, banana splits are \$1.75, and egg creams are \$.40. The portions at Steve’s are large, and so are the lines, so expect to do a little waiting. It’s worth it.” [?] pág. 224

Sintaxis

```
class unaClase
  include <nombre-módulo>
  ...
end
```

```
1  #modulo
2  module Stack
3      def push(item) end
4      def pop; end
5  end
6
7  #clase incluye modulo
8  class StackImpl
9      include Stack
10     @stackArray
11     @tos
12     public
13     def initialize
14         @stackArray = []
15         @tos = -1
16     end
17
18     #implementa el método definido en el módulo
19     def push item
20         @stackArray[@tos+=1] = item
21     end
22
23     #implementa el método definido en el módulo
24     def pop
25         objRef = @stackArray[@tos]
26         @stackArray[@tos] = nil
27         @tos-=1
28         return objRef
29     end
30
31     def peek
```

```
32         return @stackArray[@tos]
33     end
34 end
35
36 # extendiendo un módulo puede ser hecho en el mismo módulo
37 # dinámicamente o en otro módulo. Pero en este caso las queremos separadas
38
39 =begin
40 module Stack
41     def isEmpty
42     end
43     def isFull
44     end
45 end
46 =end
47
48 module SafeStack
49     def isEmpty
50     end
51     def isFull
52     end
53 end
54
55 # esta clase hereda la implementación de la pila StackImpl
56 # e implementa la nueva interfaz extendida SafeStack
57 class SafeStackImpl < StackImpl
58     include SafeStack
59
60     def initialize
61         super
62     end
63
64     # implementa los métodos de la interfaz
65     def isEmpty
66         return tos < 0
67     end
68     def isFull
69         tos >= stackArray.length
70     end
71 end
72
73 def stackUser
```

```
74     safeStackRef = SafeStackImpl.new
75
76     safeStackRef.push("Dolar");
77     safeStackRef.push "Peso"
78     puts safeStackRef.pop
79     puts safeStackRef.pop
80
81     puts safeStackRef.class
82 end
83
84 stackUser
```

Listing 171. Ejemplo de mixins en Ruby.

Aunque la instrucción *include* nos puede recordar a la misma instrucción en C/C++, en realidad es diferente. *#include* en C/C++ es ejecutada por el preprocesador e indica que el archivo correspondiente debe ser insertado momentos previos a la compilación.

Además, en Ruby, no se copia el código, sino que se hace una referencia de la clase al módulo incluido. Múltiples clases pueden tener referencia al mismo módulo y si este módulo es modificado, inclusive mientras el programa esta ejecutándose, todas las clases se verán afectadas en su comportamiento [?]. Mucha de la fuerza del uso de *mixins* esta en la funcionalidad que añade a las clases, pero en código bien diseñado este poder puede verse aún más si el código en el *mixin* interactúa con el código de las clases.

18.5 Herencia: Implementación en Python

El manejo de herencia en Python se hace de la siguiente manera¹⁰:

Sintaxis
<pre>class <ClaseDerivada>(<ClaseBase>): <instrucciones></pre>
Si la clase se encuentra en otro módulo:
<pre>class <ClaseDerivada>(<Módulo>.<ClaseBase>): <instrucciones></pre>

Todas las clases en Python 3 derivan de la clase *object*. Los métodos de la clase base pueden ser sobrescritos. Un objeto ejecutando métodos puede ser que llame a métodos de la clase base o métodos sobrescritos en la jerarquía de herencia, de manera similar a los métodos virtuales en C++.

Si fuera necesario llamar a un método de clase base que se haya sobrescrito en una clase derivada puede hacerse.

Sintaxis
<code><ClaseBase>.<método>(self, <argumentos>)</code>

Dos métodos útiles en herencia son *isinstance*(*< objeto >*, *< clase >*), el cual devuelve verdadero si un objeto pertenece a una clase o subclase especificada (directa, indirecta o virtual); de forma similar, *issubclass*(*< clase >*, *< infoclase >*) regresa verdadero si *< clase >* es una subclase (directa, indirecta o virtual) de *< infoclase >*¹¹.

Ejemplo:

```
1 class Persona:
2     def __init__(self, nombre, apellido):
3         self.nombre = nombre
```

¹⁰[Multiple Inheritance](#)

¹¹Aquí, una clase es considerada subclase de si misma.

```
4         self.apellido = apellido
5
6     def nombreCompleto(self):
7         return self.nombre + " " + self.apellido
8
9 class Empleado(Persona):
10
11     def __init__(self, nombre, apellido, staffnum):
12         Persona.__init__(self, nombre, apellido)
13         self.staffnum = staffnum
14
15     def getEmpleado(self):
16         return self.nombreCompleto() + ", " + self.staffnum
17
18 # script de ejecución inicial
19 x = Persona("Una", "Persona")
20 y = Empleado("Un", "Empleado", "121212")
21
22 print(x.nombreCompleto())
23 print(y.getEmpleado())
```

Listing 172. Ejemplo de herencia en Python.

18.5.1 Inicializadores de superclase

En Python también es posible llamar al método inicializador de la superclase usando *super* seguido de la lista de argumentos.

Ejemplo:

```
1
2 class Persona:
3     def __init__(self, nombre, edad, sexo):
4         self.nombre = nombre
5         self.edad = edad
6         self.sexo = sexo
7
8     def __str__(self):
9         return self.nombre + " " + str(self.edad) + " " + self.sexo
10
11 class Estudiante(Persona):
12     def __init__(self, nombre, edad, sexo, matr, horas):
```

```

13         # en Python 3 no se necesita pasar self al llamar
14         # a super() ni la sintaxis super(subclase, self)
15         # de Python 2
16         super().__init__(nombre, edad, sexo)
17         self.matricula = matr
18         self.horas = horas
19     def __str__(self):
20         return super().__str__() + " " + self.matricula + " " + str(self.horas)
21
22     #creando dos objetos
23     a = Persona("Ironman", 37, "m")
24     b = Estudiante("Spiderman", 36, "m", "000-13-5031", 24)
25
26     print(a)
27     print(b)

```

Listing 173. Ejemplo con inicializadores de superclase en Python.

18.5.2 Herencia Múltiple (*mixins*) en Python

Python soporta una forma limitada de herencia múltiple. Algunos autores se refieren a la implementación de Python de herencia múltiple como mixins¹². La definición de una clase con herencia múltiple tiene la siguiente forma:

Sintaxis

```

class <ClaseDerivada>(<ClaseBase1>, <ClaseBase2>, ... ):
    <instrucciones>

```

Ejemplo:

```

1 class Base1 :
2     def __init__(self, x):
3         self.value=x
4
5     def getData(self):

```

¹²[Mixins and Python](#)


```
6         return self.value
7
8     class Base2 :
9         def __init__(self, c):
10             self.letter=c
11
12         def getData(self):
13             return self.letter
14
15     # herencia múltiple
16     class Derived (Base1, Base2) :
17
18         def __init__(self, i, c, f):
19             Base1.__init__(self, i)
20             Base2.__init__(self, c)
21             self.real=f
22
23         def getReal(self) :
24             return self.real
25
26     #script de inicio de ejecución
27     b1 = Base1(10)
28     b2 = Base2('Z')
29     d = Derived(7, 'A', 3.5)
30
31     print("Objeto b1 contiene entero ", b1.getData())
32     print("Objeto b2 contiene caracter ", b2.getData())
33     print("Objeto d contiene ", d.getData())
```

Listing 174. Ejemplo de mixins en Python.

En Python, la jerarquía de herencia es definida de derecha a izquierda por lo que hay que tener en cuenta que el resultado puede variar si una clase de más a la izquierda sobrescribe métodos o atributos. Por eso se considera a la clase de la extrema derecha como la “clase base principal”, mientras que las otras clases agregan características (es por este aspecto por el que se maneja a veces como mixins). *En caso de sobrescribir un método, se ejecutará el que se encuentre más a la extrema izquierda (el último en ser sobrescrito)*

Tener en cuenta que la llamada a múltiples inicializadores de clase en herencia múltiple si puede en la práctica generar conflictos. Lo sugerido es en caso de herencia múltiple siempre hacer llamadas a inicializadores de clase base mediante el nombre de la clase base (en lugar de super) **a través de toda la jerarquía de herencia**, no únicamente en la clase que tiene herencia múltiple

Problema: Firmas de Métodos Incompatibles

Si cada `__init__` tiene parámetros fijos, el sistema se rompe. `super()` llama al siguiente método en el MRO, pero no sabe qué parámetros espera ese método.

```

1  # EJEMPLO INCORRECTO - ESTO DARÁ UN ERROR
2  class Motor:
3      def __init__(self, cilindros):
4          self.cilindros = cilindros
5
6  class Chasis:
7      def __init__(self, material):
8          self.material = material
9
10 class Coche(Motor, Chasis):
11     def __init__(self, cilindros, material, color):
12         # ¿Cómo llamamos a super? ¿Con qué parámetros?
13         # super().__init__(cilindros) # Falla porque Chasis no espera 'cilindros'
14         # super().__init__(material) # Falla porque Motor no espera 'material'
15         self.color = color

```

Este código está destinado a fallar porque no hay una forma de que `super().__init__` satisfaga a *Motor* y a *Chasis* al mismo tiempo.

La Solución Cooperativa con ****kwargs**

Para solucionarlo, modificamos todos los inicializadores para que acepten

`**kwargs`

. Cada método consume los argumentos que necesita del diccionario `kwargs` y pasa el resto al siguiente nivel.

Veamos el ejemplo de una jerarquía con parámetros específicos para cada clase.

Sintaxis

```

class SerVivo:
    def __init__(self, *, nombre, **kwargs):
        print(f"SerVivo necesita 'nombre'. Recibe: {nombre}")
        super().__init__(**kwargs) # Pasa los kwargs restantes
        self.nombre = nombre

class Ave(SerVivo):
    def __init__(self, *, envergadura, **kwargs):
        print(f"Ave necesita 'envergadura'. Recibe: {envergadura}")
        super().__init__(**kwargs) # Pasa los kwargs restantes
        self.envergadura = envergadura

class Mamifero(SerVivo):
    def __init__(self, *, tipo_pelaje, **kwargs):
        print(f"Mamifero necesita 'tipo_pelaje'. Recibe: {tipo_pelaje}")
        super().__init__(**kwargs) # Pasa los kwargs restantes
        self.tipo_pelaje = tipo_pelaje

class Murcielago(Ave, Mamifero):
    def __init__(self, *, habilidad_ecolocalizacion, **kwargs):
        print(f"Murcielago necesita 'habilidad_ecolocalizacion'. Recibe: {habilidad_eco
        super().__init__(**kwargs) # Pasa los kwargs restantes
        self.habilidad_ecolocalizacion = habilidad_ecolocalizacion

# --- Probemoslo ---
print("Creando un murciélago con todos los parámetros...")
# Al crear el objeto, debemos proveer TODOS los parámetros que la cadena necesita.
mi_murcielago = Murcielago(
    nombre="Batty",
    tipo_pelaje="Corto",
    envergadura=0.3,
    habilidad_ecolocalizacion=True
)

# Verificamos que todos los atributos se asignaron correctamente
print("\n--- Atributos del Murciélago ---")
print(f"Nombre: {mi_murcielago.nombre}")
print(f"Pelaje: {mi_murcielago.tipo_pelaje}")
print(f"Envergadura: {mi_murcielago.envergadura}m")
print(f"Ecolocalización: {mi_murcielago.habilidad_ecolocalizacion}")

```

¿Cómo funcionó?

- *Murcielago*(*nombre* = ..., *tipo_pelaje* = ..., ...): Se llama al inicializador de *Murcielago* con un diccionario de 4 argumentos.
- *Murcielago*.*__init__*: Toma su argumento *habilidad_ecolocalizacion*. Llama a *super().__init__* pasándole los 3 argumentos restantes (*nombre*, *tipo_pelaje*, *envergadura*).
- *Ave*.*__init__*(*siguiente en el MRO*): Recibe los 3 argumentos. Toma *envergadura* y llama a *super().__init__* con los 2 restantes (*nombre*, *tipo_pelaje*).
- *Mamifero*.*__init__*(*siguiente en el MRO*): Recibe los 2 argumentos. Toma *tipo_pelaje* y llama a *super* (último en el MRO con *__init__* personalizado): Recibe el argumento *nombre*. Lo toma y llama a *super().__init__()* sin argumentos.
- *object*.*__init__*: Finalmente, se llama al inicializador de la clase *object* base, que no toma argumentos, y la cadena termina limpiamente.

El uso de ¹³* en la definición (*def __init__(self, *, nombre, **kwargs)*) fuerza a que *nombre* sea un argumento de palabra clave (*keyword-only argument*), lo cual hace el código más robusto y legible.

Simplicidad Aparente vs. Robustez Real

Al trabajar con herencia múltiple en Python, surge la pregunta de cómo inicializar las clases base desde una clase derivada. A primera vista, llamar directamente al inicializador de cada clase padre, como *Base1.__init__(self, ...)*, puede parecer más simple y directo. Sin embargo, esta simplicidad es engañosa y, a menudo, conduce a un código más frágil y difícil de mantener, especialmente en sistemas complejos.

El enfoque de llamada directa es **imperativo**: le indicamos a Python exactamente qué hacer y en qué orden. En contraste, el enfoque con *super()* es **cooperativo**: cada clase en la jerarquía confía en la siguiente de la cadena para que realice su parte correspondiente.

18.6 Comparativa de Enfoques

La siguiente tabla resume las diferencias fundamentales entre ambos métodos:

¹³* por sí solo sirve para indicar que todos los parámetros que vienen después de él deben ser especificados usando su nombre (como argumentos de palabra clave) y no por su posición. Pensemos en el * como una barrera que separa los argumentos posicionales (a la izquierda) de los argumentos que son exclusivamente de palabra clave (a la derecha).

Característica	Llamada Directa (Base.__init__)	super () Cooperativo
Simplicidad Inicial	Alta. Es muy obvio lo que se está llamando.	Moderada. Requiere entender el patrón <code>**kwargs</code> y el MRO (Method Resolution Order).
Problema del Diamante	No lo resuelve . Causa ejecuciones duplicadas del inicializador <code>.abuelo</code> ".	Lo resuelve . Garantiza que cada inicializador en la jerarquía se ejecute solo una vez.
Mantenibilidad	Muy Baja. Si la jerarquía de clases cambia, se debe reescribir el <code>__init__</code> de la clase hija. El código es frágil.	Muy Alta. La jerarquía puede cambiar. Mientras se respeta el patrón <code>super ()</code> , el código se adapta automáticamente.
Acoplamiento	Alto. La clase hija necesita conocer toda la estructura de herencia de sus padres para pasar los parámetros correctos.	Bajo. Cada clase solo necesita saber de sí misma. Toma sus argumentos y pasa el resto. No le importa quién es el siguiente.

Cuadro 18.1. Comparación entre llamada directa y `super ()` cooperativo.

18.7 El Problema de la Responsabilidad

Consideremos una jerarquía de clases donde `Murcielago` hereda de `Ave` y `Mamifero`, y ambas heredan de `SerVivo`. Si usamos la llamada directa, la clase `Murcielago` asume demasiada responsabilidad.

18.7.1 Enfoque Frágil con Llamada Directa

```

1 class Murcielago(Ave, Mamifero):
2     def __init__(self, nombre, tipo_pelaje, envergadura, habilidad_ecolocalizacion):
3         # La clase Murcielago debe saber qué parámetros necesita cada padre.
4         Ave.__init__(self, nombre=nombre, envergadura=envergadura)
5         Mamifero.__init__(self, nombre=nombre, tipo_pelaje=tipo_pelaje)
6
7         self.habilidad_ecolocalizacion = habilidad_ecolocalizacion

```

Los problemas con este enfoque son:

1. **Conocimiento Excesivo:** Murcielago está forzado a saber que tanto Ave como Mamifero necesitan el parámetro nombre para pasárselo a SerVivo. Si la estructura de herencia cambia, el código se rompe.
2. **Duplicación de Lógica:** El método SerVivo.__init__ se ejecuta dos veces, lo cual es ineficiente y puede causar errores.

Enfoque Robusto con super ()

Con `super()`, la responsabilidad se distribuye correctamente, siguiendo el principio de bajo acoplamiento.

```
1 class Murcielago(Ave, Mamifero):
2     def __init__(self, **kwargs):
3         # Murcielago solo se preocupa por sus propios asuntos.
4         # Pasa el resto de responsabilidades al siguiente en la cadena.
5         super().__init__(**kwargs)
6         # ...lógica específica de Murciélago...
```

Este diseño sigue un principio clave de la POO: una clase no debería necesitar conocer los detalles internos de la implementación de sus padres.

Especificar cada inicializador es como darle a un taxista una lista de instrucciones detalladas para un viaje. Funciona para esa ruta específica, pero falla ante cualquier imprevisto. Usar `super()` es como darle la dirección final; el conductor (el MRO de Python) sabe cómo navegar de la manera más eficiente.

Aunque la llamada directa **parece más simple**, es una simplicidad que crea código rígido y propenso a errores. El modelo cooperativo de `super()` es la base para construir sistemas flexibles y mantenibles en Python.

Comentarios finales sobre herencia múltiple en Python

1. **Inicialización redundante:** En casos de herencia múltiple, atributos como 'nombre' y 'edad' pueden inicializarse más de una vez. Por ejemplo, en la clase 'EstProf', estos atributos se inicializan tanto en 'EstudiantePosgrado' como en 'Profesor'. Aunque esto no genera problemas en este caso porque los valores son idénticos, es importante tener cuidado en jerarquías más complejas donde los valores podrían diferir o causar inconsistencias.
2. **Orden de resolución de métodos (MRO):** Python utiliza un mecanismo llamado *Method Resolution Order* (MRO) para determinar el orden en el que busca métodos y atributos en la jerarquía de clases. Puedes verificar el MRO de una clase ejecutando el siguiente comando: `< clase > .mro()`. Esto mostrará el orden en el que Python recorrerá las clases cuando se invoquen métodos o inicializadores.

3. **Evitar *super()* en herencia diamante.** En estructuras de herencia diamante (donde una clase hereda de dos clases que comparten un ancestro común), el uso de *super()* puede no ser la mejor opción, ya que podría omitir la inicialización completa de ciertas clases. En estos casos, es más confiable llamar explícitamente a los inicializadores de los padres involucrados para asegurar que todos los atributos y métodos se configuren correctamente. Sin embargo, llamar directamente a los inicializadores de clases base, puede llevar a la redundancia de inicialización de atributos y además a posibles errores de inicialización. En este caso se sugiere **rediseñar la inicialización** para que los atributos compartidos solo se inicialicen una vez, preferiblemente en la clase base más general. Las clases derivadas pueden encargarse de inicializar únicamente los atributos específicos que les corresponden.

18.8 Herencia: Implementación en Scala

En su forma más simple, la herencia en Scala se maneja de una forma bastante similar a Java¹⁴.

Sintaxis

```
class <Derivada> extends <Base> {  
}
```

Solamente recordar que es posible definir los atributos como parámetro de las clases -polimórfica- (base y derivada) definiendo al mismo tiempo el constructor primario.

18.8.1 Inicializadores de superclase

La inicialización de superclase se hace en el momento de definir la herencia en la subclase. Los constructores auxiliares no pueden invocar directamente a constructores de la superclase; sólo pueden invocar al constructor primario de su clase quien a su vez invocará al constructor de la superclase.

Ejemplo:

```
1 //ejemplo de herencia  
2 class Vehiculo(var ruedas: Int, var pasajeros: Int){  
3  
4     def setRuedas(num: Int){  
5         ruedas=num  
6     }  
7  
8     def getRuedas(): Int= ruedas  
9  
10    def setPasajeros(num: Int){  
11        pasajeros=num  
12    }  
13  
14    def getPasajeros(): Int={
```

¹⁴[Inheritance and Overriding in Scala](#)


```
15         return pasajeros
16     }
17 }
18
19 //clase Camion con herencia de Vehiculo
20 class Camion(ruedas: Int, pasajeros: Int, var carga: Int)
21     extends Vehiculo(ruedas, pasajeros) {
22
23     def this() = this(0, 0, 0)
24
25     def setCarga(num: Int) {
26         carga=num
27     }
28
29     def getCarga(): Int = carga
30
31     def muestra() {
32         // uso de métodos heredados
33         println("Ruedas: " + getRuedas());
34         println("Pasajeros: " + getPasajeros());
35         // método de la clase Camion
36         println("Capacidad de carga: " + getCarga());
37     }
38 }
39
40 object Main extends App {
41     val ford: Camion = new Camion()
42     ford.setRuedas(6)
43     ford.setPasajeros(3)
44     ford.setCarga(3200)
45     ford.muestra()
46 }
```

Listing 175. Ejemplo.

18.8.2 Sobreescritura de métodos

Comparado con Java, Scala añade unas restricciones para sobrecribir un método. El método sobreescrito debe usar el modificador *override* (existe una anotación *@Override* en Java, pero no es obligatoria). Adicionalmente, no se puede sobrecribir un *var* con *val* o *def*. Pero puede sobrecribirse si en la superclase es abstracta. Un *val* no puede ser sobreescrito por un *var* o *def* en una subclase.

Ejemplo:

```
1 class Elemento(var precio:Double, var descripcion:String){
2     //Sin descuento para el elemento
3     def precioDescuento():Double = precio
4
5     override def toString():String = {
6         descripcion+" Costo: "+precioDescuento()
7     }
8 }
9
10 class ElementoEspecial(precio:Double, descripcion:String,
11     var porcentajeDescuento:Double)
12     extends Elemento(precio, descripcion){
13
14     //Precio con descuento
15     override def precioDescuento():Double = {
16         precio - ((porcentajeDescuento/100) * precio)
17     }
18 }
19
20 object Main extends App{
21     val elem = new Elemento(335.65,"Libro,")
22     println(elem)
23
24     val elementoEspecialspecial = new ElementoEspecial(456.7,
25         "Un libro especial", 10)
26     println(elementoEspecialspecial)
27
28 }
```

Listing 176. Ejemplo de sobreescritura de métodos en Scala.

Ejemplo:

```
1 class Empleado(var firstName: String, var lastName: String){
2     override def toString():String =
3         firstName+" "+lastName
4
5     def print() {
6         println(toString())
7     }
8 }
```

```
9
10 class TrabajadorPorHora(firstName: String, lastName: String,
11     var wage: Float, var hours: Float)
12     extends Empleado(firstName, lastName){
13
14     def getPay(): Float = wage * hours
15
16     override def print() {
17         println("Metodo print de Trabajador por hora")
18         super.print() // llama a función de clase base
19         println(" es un trabajador por hora con sueldo de" + " $" + getPay())
20     }
21 }
22
23 object Main extends App{
24     val e = new Empleado("nombre", "apellido")
25     val h = new TrabajadorPorHora ("Juanito", "Perez", 40.0f, 7.50f)
26
27     e.print()
28     h.print
29 }
```

Listing 177. Ejemplo 2 de sobreescritura de métodos en Scala.

18.8.3 Traits & Mixins en Scala

Los *traits* (rasgos), son un tipo abstracto utilizado en OOP. Un trait representa un conjunto de métodos que son usados para extender la funcionalidad de una clase. Los *traits* son similares a los *mixins*, y también pueden incluir definiciones para métodos de clase. Los *traits* son una forma simple para reuso de comportamiento para los componentes de software. A diferencia de Java, Scala permite tener *traits* parcialmente implementados (tener implementaciones para algunos métodos). Los *traits* no pueden instanciarse y no tienen parámetros de constructor.

Sintaxis

```

trait <nombre> {
  def <método 1>(<parámetros>) : <tipo> [<implementación>]
  def <método 2>(<parámetros>) : <tipo> [<implementación>]
  ...
  def <método n>(<parámetros>) : <tipo> [<implementación>]
}

```

Las clases que integren *traits* deben implementar los métodos que no se encuentren implementados. Los métodos implementados se heredan directamente.

La clase *Any* es la clase raíz de la jerarquía en Scala¹⁵.

Ejemplo:

```

1 trait Similar {
2   def esSimilar(x: Any): Boolean
3   def noEsSimilar(x: Any): Boolean = !esSimilar(x)
4 }
5
6 class Punto(xc: Int, yc: Int) extends Similar {
7   var x: Int = xc
8   var y: Int = yc
9   def esSimilar(obj: Any) =
10     obj.isInstanceOf[Punto] &&
11     obj.asInstanceOf[Punto].x == x
12 }
13
14 object Main extends App {
15   val p1 = new Punto(2, 3)
16   val p2 = new Punto(2, 4)
17   val p3 = new Punto(3, 3)
18   println(p1.noEsSimilar(p2))
19   println(p1.noEsSimilar(p3))
20   println(p1.noEsSimilar(2))
21 }

```

Listing 178. Ejemplo de mixins en Scala.

¹⁵[Class Any](#)

Una clase puede usar *extends* para heredar un *trait* sólo si no hereda ya de una clase base. Si ya se hereda de una clase o se quiere heredar más *traits* se usa la palabra clave *with*.

Ejemplo:

```
1 class Basel(x: Int) {
2     var value: Int = x
3     //conflicto de miembros en clase derivada
4     //def getData() = value
5     def getValue() = value
6 }
7
8 trait MiTrait {
9     var letter: Char
10    def setData(c: Char)
11    def getData(): Char = letter
12 }
13
14 class Derivada(x:Int, c:Char, r: Float) extends Basel(x) with MiTrait {
15     var real: Float = r
16     var letter= c
17     def setData(c: Char) {
18         letter=c
19     }
20     def getReal(): Float = real
21 }
22
23 object Main extends App {
24     val b1 = new Basel(10)
25     val d = new Derivada(7, 'A', 3.5f)
26     println("Objeto b1 contiene entero " + b1.getValue())
27     println("Objeto d contiene " + d.getData())
28 }
```

Listing 179. Ejemplo mixins usando *with* en Scala.

18.9 Herencia: Implementación en C#

C# permite la herencia simple entre clases heredando de una clase base a una clase derivada.

La sintaxis de la herencia en C# es similar a la implementación básica en C++.

Sintaxis

```
class <claseDerivada> : <claseBase> {  
    //cuerpo subclase  
}
```

Ejemplo:

```
1  using System;  
2  
3  //ejemplo de herencia  
4  public class Vehiculo{  
5      private int ruedas;  
6      private int pasajeros;  
7  
8      public void setRuedas (int num) {  
9          ruedas=num;  
10     }  
11  
12     public int getRuedas () {  
13         return ruedas;  
14     }  
15  
16     public void setPasajeros (int num) {  
17         pasajeros=num;  
18     }  
19  
20     public int getPasajeros () {  
21         return pasajeros;  
22     }  
23 }  
24
```

```
25 //clase Camion con herencia de Vehiculo
26 public class Camion : Vehiculo {
27     private int carga;
28
29     public void setCarga(int num) {
30         carga=num;
31     }
32
33     public int getCarga() {
34         return carga;
35     }
36
37     public void muestra() {
38         // uso de métodos heredados
39         Console.WriteLine("Ruedas: " + getRuedas());
40         Console.WriteLine("Pasajeros: " + getPasajeros());
41         // método de la clase Camion
42         Console.WriteLine("Capacidad de carga: " + getCarga());
43     }
44
45     public static void Main() {
46         Camion ford= new Camion();
47         ford.setRuedas(6);
48         ford.setPasajeros(3);
49         ford.setCarga(3200);
50         ford.muestra();
51     }
52 }
```

Listing 180. Ejemplo de herencia en C#.

18.9.1 Acceso a miembros

Un miembro privado de una clase permanecerá privado a su clase. No será accesible fuera de la clase, incluidas sus clases derivadas. Un miembro privado tendrá que ser accedido mediante un método público o protegido.

18.9.2 Constructores de clase base

Como en otros lenguajes, el constructor de la clase base puede ser llamado desde la clase derivada, para inicializar los atributos heredados.

Sintaxis

```
<constructor>(<lista de argumentos>) : base(<argumentos sin el tipo>)
```

18.9.3 Interfaces en C#

C# cuenta con manejo de interfaces de una manera similar a Java.

Una interfaz tiene un formato muy similar a una clase, sus principales características:

- Una interfaz proporciona los nombres de los métodos, pero no sus implementaciones.
- Una clase puede implementar varias interfaces, aunque solo pueda heredar una clase.
- No es posible crear instancias de una interfaz.
- La clase que implementa la interfaz debe escribir el código de todos los métodos, de otra forma no se podrá generar instancias de esa clase.

El formato general para la declaración de una interfaz es el siguiente:

Sintaxis

```
[public] interface <nombreInterfaz> {  
    //descripción de miembros  
    //los métodos no incluyen código:  
    <acceso> <tipo> <nombreMetodo> ( <parámetros> ) ;  
}
```

El cuerpo de la interfaz generalmente es una lista de prototipos de métodos, pero puede contener atributos si se requiere.

Una clase implementa una o más interfaces de la siguiente forma:

Sintaxis

```
class <claseDerivada> : <claseBase> ,    <nombreInterfaz> [, ... <interfazN>] {
    //definición de la clase
    //debe incluirse la definición de los métodos de la interfaz
    //con la implementación del código de dichos métodos.
}
```

Además, una interfaz puede ser extendida de la misma forma que una clase, aprovechando las interfaces previamente definidas:

Sintaxis

```
[public] interface <nombreInterfaz> : <InterfazBase> {
    //descripción de miembros
}
```

Ejemplo:

```
1  using System;
2
3  //interfaz
4  interface IStack {
5      void push(Object item);
6      Object pop();
7  }
8
9  //clase implementa la interfaz
10 class StackImpl : IStack {
11     protected Object[] stackArray;
12     protected int tos;
13
14     public StackImpl(int capacity) {
15         stackArray = new Object[capacity];
```

```
16         tos = -1;
17     }
18
19     //implementa el método definido en la interfaz
20     public void push(Object item)
21     { stackArray[++tos] = item; }
22
23     //implementa el método definido en la interfaz
24     public Object pop() {
25         Object objRef = stackArray[tos];
26         stackArray[tos] = null;
27         tos--;
28         return objRef;
29     }
30
31     public Object peek() { return stackArray[tos]; }
32 }
33
34 // extendiendo una interfaz
35 interface ISafeStack : IStack {
36     bool isEmpty();
37     bool isFull();
38 }
39
40 //esta clase hereda la implementación de la pila StackImpl
41 // e implementa la nueva interfaz extendida ISafeStack
42 class SafeStackImpl : StackImpl, ISafeStack {
43
44     public SafeStackImpl(int capacity):base(capacity) { }
45
46     //implementa los métodos de la interfaz
47     public bool isEmpty() { return tos < 0; }
48     public bool isFull() { return tos >= stackArray.Length; }
49 }
50
51
52 public class StackUser {
53
54     public static void Main() {
55         SafeStackImpl safeStackRef = new SafeStackImpl(10);
56         StackImpl stackRef = safeStackRef;
57         ISafeStack isafeStackRef = safeStackRef;
```

```
58         IStack istackRef = safeStackRef;
59         Object objRef = safeStackRef;
60
61         safeStackRef.push("Dolar");
62         stackRef.push("Peso");
63         Console.WriteLine(isafeStackRef.pop());
64         Console.WriteLine(istackRef.pop());
65
66         Console.WriteLine(objRef.GetType());
67     }
68 }
```

Listing 181. Ejemplo de interfaces en C#.

18.10 Herencia: Implementación en D

D permite la herencia simple entre clases heredando de una clase base a una clase derivada.

Así como en C#, la sintaxis de la herencia en D es similar a la implementación básica en C++.

Como en otros lenguajes (e.g., Java, C#), todas las clases heredan directa o indirectamente de la clase *Object*.

Sintaxis

```
class <claseDerivada> : <claseBase> {  
    //cuerpo claseDerivada  
}
```

Ejemplo:

```
1 module Vehiculo;  
2  
3 import std.stdio;  
4  
5 //ejemplo de herencia  
6 class Vehiculo{  
7     private:  
8         int ruedas;  
9         int pasajeros;  
10  
11     public:  
12         void setRuedas (int num) {  
13             ruedas=num;  
14         }  
15  
16         int getRuedas () {  
17             return ruedas;  
18         }  
19  
20         void setPasajeros (int num) {  
21             pasajeros=num;
```

```
22     }
23
24     int getPasajeros() {
25         return pasajeros;
26     }
27 }
28
29 //clase Camion con herencia de Vehiculo
30 class Camion : Vehiculo {
31     private:
32     int carga;
33
34     public:
35     void setCarga(int num) {
36         carga=num;
37     }
38
39     int getCarga() {
40         return carga;
41     }
42
43     void muestra() {
44         // uso de metodos heredados
45         writeln("Ruedas: ", getRuedas());
46         writeln("Pasajeros: ", getPasajeros());
47         // metodo de la clase Camion
48         writeln("Capacidad de carga: ", getCarga());
49     }
50 }
51
52 void main() {
53     Camion ford= new Camion();
54     ford.setRuedas(6);
55     ford.setPasajeros(3);
56     ford.setCarga(3200);
57     ford.muestra();
58 }
```

Listing 182. Ejemplo de herencia en D.

18.10.1 Acceso a miembros

En D existen los modificadores de acceso *private*, *package*, *protected*, *public* y *export*¹⁶.

- *private*. Un miembro privado implica que únicamente miembros de la clase o miembros y funciones del módulo de la clase pueden acceder a ellos. Miembros privados no pueden ser sobrescritos.
- *package*. El modificador *package* extiende la privacidad de manera que estos miembros pueden ser accedidos por otros módulos en el mismo paquete.
- *protected*. Un miembro protegido miembros de su clase, clase derivada o miembros y funciones en el mismo módulo pueden acceder a dicho miembro.
- *public*. Un miembro público puede ser accedido donde el miembro sea visible.
- *export*. El modificador *export* implica que cualquier código fuera del ejecutable al que pertenece puede acceder a éste. Es análogo a la definición de exportación para DLLs.

18.10.2 Constructores de clase base

Como en otros lenguajes, el constructor de la clase base puede ser llamado desde la clase derivada, para inicializar los atributos heredados.

Sintaxis

```
this<lista de argumentos> {  
    ...  
    super(<lista de argumentos sin el tipo>)  
    ...  
}
```

18.10.3 Interfaces en D

D cuenta con manejo de interfaces de una manera similar a Java y C#.

Una interfaz tiene un formato muy similar a una clase, sus principales características:

¹⁶Classes, <http://dlang.org/spec/class.html>

- Una interfaz proporciona los nombres de los métodos, pero no sus implementaciones.
- Una clase puede implementar varias interfaces, aunque solo pueda heredar una clase.
- No es posible crear instancias de una interfaz.
- Métodos **estáticos** y **finales** deben llevar su implementación. Las clases no pueden sobrescribir estos tipos de miembros.
- La clase que implementa la interfaz debe escribir el código de todos los métodos, de otra forma no se generará un error.
- Una interfaz no puede tener atributos.

El formato general para la declaración de una interfaz es el siguiente¹⁷:

Sintaxis

```
interface <nombreInterfaz> {  
    //descripción de miembros  
    //los métodos no incluyen código:  
    <acceso> <tipo> <nombreMetodo> ( <parámetros> ) ;  
}
```

Una clase hereda una interfaz a través de la misma forma que hereda de una clase:

Sintaxis

```
class <clase> : <nombreInterfaz> {  
    //definición de la clase  
    //debe incluirse la definición de los métodos de la interfaz  
    //con la implementación del código de dichos métodos.  
}
```

¹⁷,Interface

De hecho, si la clase hereda una previa implementación de una interfaz pero también hereda la interfaz directamente, esta clase esta obligada a reimplementar la interfaz.

Ejemplo:

```

1 interface D {
2     int foo();
3 }
4
5 class A : D {
6     int foo() { return 1; }
7 }
8
9 class B : A, D{
10 }    // error, no se reimplementó foo() para la interfaz D

```

Además, una interfaz puede ser extendida de la misma forma que una clase, aprovechando las interfaces previamente definidas.

Sintaxis

```

interface <nombreInterfaz> : <InterfazBase> {
    //descripción de miembros
}

```

Ejemplo:

```

1 module Stack;
2 import std.stdio;
3
4 //interfaz
5 interface IStack {
6     void push(Object item);
7     Object pop();
8 }
9
10 //clase implementa la interfaz
11 class StackImpl : IStack {

```



```

12     protected:
13         Object[] stackArray;
14     int tos;
15     public:
16     this(int capacity) {
17         stackArray = new Object[capacity];
18         tos = -1;
19     }
20
21     //implementa el metodo definido en la interfaz
22     void push(Object item)
23     { stackArray[++tos] = item; }
24
25     //implementa el metodo definido en la interfaz
26     Object pop() {
27         Object objRef = stackArray[tos];
28         stackArray[tos] = null;
29         tos--;
30         return objRef;
31     }
32
33     Object peek() { return stackArray[tos];}
34 }
35
36 // extendiendo una interfaz
37 interface ISafeStack : IStack {
38     bool isEmpty();
39     bool isFull();
40 }
41
42 //esta clase hereda la implementacion de la pila StackImpl
43 // e implementa la nueva interfaz extendida ISafeStack
44 class SafeStackImpl : StackImpl, ISafeStack {
45     public:
46     this(int capacity){super(capacity); }
47     //implementa los metodos de la interfaz
48     bool isEmpty() { return tos < 0; }
49     bool isFull() { return tos >= stackArray.length;}
50 }
51
52 class String{
53     public:

```

```
54     string myString;
55     this(string s) {
56         myString=s;
57     }
58 }
59
60 void main() {
61     SafeStackImpl safeStackRef = new SafeStackImpl(10);
62     StackImpl stackRef = safeStackRef;
63     ISafeStack isafeStackRef = safeStackRef;
64     IStack istackRef = safeStackRef;
65     Object objRef = safeStackRef;
66     //safeStackRef.push("dolar"); // strings no son objetos
67     safeStackRef.push(new String("dolar"));
68     stackRef.push(new String("Peso"));
69     String s = cast(String) isafeStackRef.pop();
70     writeln(s.myString);
71     s = cast(String) istackRef.pop();
72     writeln(s.myString);
73     writeln(objRef.toString());
74 }
```

Listing 183. Ejemplo de interfaces en D.

18.10.4 Traits y Mixins en D

Traits y *mixins* no van por el sentido del uso de la jerarquía de herencia, pero los mencionamos aquí para evitar confusión al respecto.

Los *traits* en D son extensiones para permitir que los programas, en tiempo de compilación, obtengan información general para el compilador. Esto es también conocido como reflexión (*reflection*) en tiempo de compilación.

Los *mixins* en D permiten a unas cadenas constantes ser compiladas como código regular e insertado en el programa. De manera superficial, el concepto es similar a las propiedades que realiza el preprocesador de C.

Capítulo 19

Asociaciones entre clases

19.1 Introducción

Una clase puede estar relacionada con otra clase, o en la práctica un objeto con otro objeto.

Concepto
En el modelado de objetos a la relación entre clases se le conoce como asociación ; mientras que a la relación entre objetos se llama instancia de una asociación.

Ejemplo:

Una clase *Estudiante* está asociada con una clase *Universidad*. Una asociación es una **conexión** física o conceptual entre objetos. Las relaciones¹ se consideran de naturaleza **bidireccional**; es decir, ambos lados de la asociación tienen acceso a clase del otro lado de la asociación. Sin embargo, algunas veces únicamente es necesaria una asociación en una dirección (unidireccional).

Comúnmente las asociaciones se representan en los lenguajes de programación orientados a objetos como apuntadores o referencias. Donde un apuntador a una clase B en una clase A indicaría la asociación que tiene A con B; aunque no así la asociación de B con A.

Para una asociación bidireccional es necesario al menos un par de apuntadores, uno en cada clase. Para una asociación unidireccional basta un solo apuntador en la clase que mantiene la referencia.

¹El término de relación es usado muchas veces como sinónimo de asociación, debido a que el concepto surge de las relaciones en bases de datos relacionales. Sin embargo el término más apropiado es el de asociación, ya que existen en objetos otros tipos de relaciones, como la relación de agregación y la de herencia.

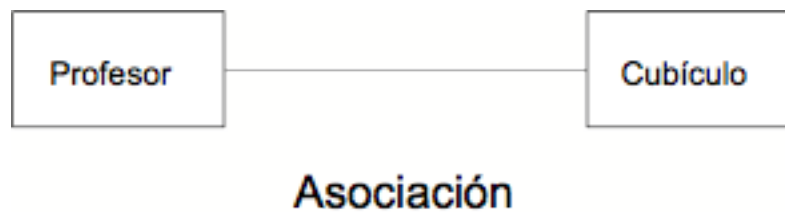


Fig. 19.1. Ejemplo de asociación en UML

En el caso de las asociaciones se asumirá que cada objeto puede seguir existiendo de manera independiente, a menos que haya sido creado por el objeto de la clase asociada, en cuyo caso deberá ser eliminado por el destructor del objeto que la creó. Es decir:

Explicación
Si el objeto A crea al objeto B , es responsabilidad de A eliminar a la instancia B antes de que A sea eliminada. En caso contrario, si B es independiente de la instancia A , A debería enviar un mensaje al objeto B para que asigne <i>NULL</i> al apuntador de B o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida.

Es importante señalar que las medidas que se tomen pueden variar de acuerdo a las necesidades de la aplicación, pero bajo ningún motivo se deben dejar accesos a áreas de memoria no permitidas o dejar objetos "volando", sin que nadie haga referencia a ellos.

Mencionamos a continuación estructuras clásicas que pueden ser vistas como una asociación:

1. Ejemplo de asociación **unidireccional**: lista ligada.
2. Ejemplo de asociación **bidireccional**: lista doblemente ligada.

19.2 Asociaciones reflexivas

Es posible tener un tipo de asociación conocida como asociación **reflexiva**.

Concepto
Si una clase mantiene una asociación consigo misma se dice que es una asociación reflexiva .

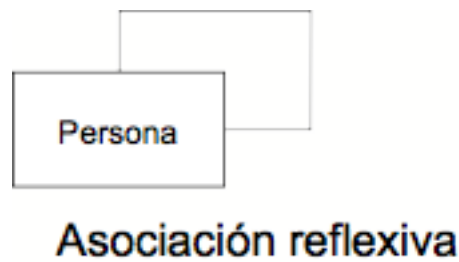


Fig. 19.2. Asociación reflexiva

Ejemplo: *Persona* puede tener asociaciones entre sí, si lo que nos interesa es representar a las personas que guardan una relación entre sí, por ejemplo si son parientes. Es decir, un objeto mantiene una asociación con otro objeto de la misma clase.

En términos de implementación significa que la clase tiene una referencia a si misma. De nuevo podemos poner de ejemplo a la clase *Nodo* en una lista ligada.

19.3 Multiplicidad de una asociación

La **multiplicidad** de una asociación especifica cuantas instancias de una clase se pueden asociar a una sola instancia de otra clase.

Se debe determinar la multiplicidad para cada clase en una asociación.

19.3.1 Tipos de asociaciones según su multiplicidad

"uno a uno": donde dos objetos se asocian de forma exclusiva, uno con el otro. Ejemplo: Uno: Un alumno tiene una boleta de calificaciones. Uno: Una boleta de calificaciones pertenece a un alumno.

"uno a muchos": donde uno de los objetos puede estar asociado con muchos otros objetos. Ejemplo: Uno: un libro solo puede estar prestado a un alumno. Muchos: Un usuario de la biblioteca puede tener muchos libros prestados.

"muchos a muchos": donde cada objeto de cada clase puede estar asociado con muchos otros objetos. Ejemplo: Muchos: Un libro puede tener varios autores. Muchos: Un autor puede tener varios libros.

Podemos apreciar en un diagrama las diversas multiplicidades:

Finalmente, es importante señalar que el control de las asociaciones no se encuentra en general apoyado por los lenguajes de programación, a pesar de ser una necesidad natural en el modelado orientado a objetos, por lo que toda la responsabilidad recae sobre el programador.

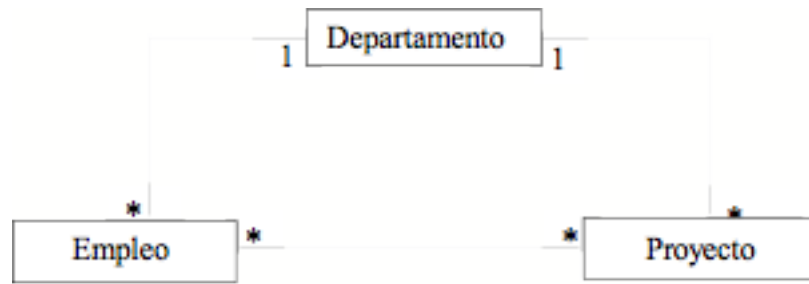


Fig. 19.3. Ejemplo de multiplicidad

19.4 Asociaciones en C++

Ejemplo: un programa que guarda una asociación bidireccional entre clases A y B.

```

1  class A{
2      //lista de atributos
3
4      B      *pB;
5  };
6
7  class B{
8      //lista de atributos
9      A      *pA;
10 };

```

En el ejemplo anterior se presenta una asociación bidireccional, por lo que cada clase tiene su respectivo apuntador a la clase contraria de la asociación. Además, deben proporcionarse métodos de acceso a la clase asociada por medio del apuntador.

19.4.1 Multiplicidad de una asociacion en C++

La forma de implementar en C++ este tipo de relaciones puede variar, pero la más común es por medio de apuntadores a objetos. Suponiendo que tenemos asociaciones bidireccionales:

- **"uno a uno"**. Un apuntador de cada lado de la asociación, como se ha visto anteriormente.
- **"uno a muchos"**. Un apuntador de un lado y un arreglo de apuntadores a objetos definido dinámica o estáticamente.

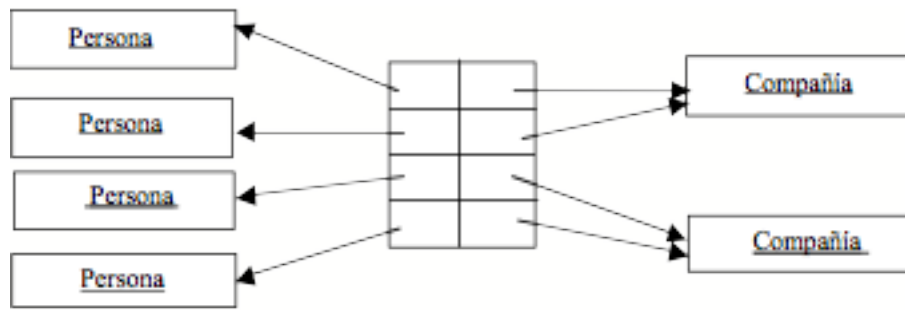


Fig. 19.4. Implementación de multiplicidad mediante tabla de referencias

```

1  class A{
2      ...
3      B *pB;
4  };
5
6  class B{
7      A *p[5];
8      //ó
9      A **p;
10 }

```

Otra forma es manejar una clase que agrupe a pares de direcciones en un objeto independiente de la clase. Por ejemplo una lista o tabla de referencias.

- **“muchos a muchos”**. Normalmente se utiliza un objeto u objetos independientes que mantiene las asociaciones entre los objetos, de manera similar a la gráfica anterior.

Ejemplo: Se muestra un código simplificado para manejo de asociaciones.

Clase Libro

```

1  #ifndef LIBRO_H_
2  #define LIBRO_H_
3
4  class Persona;
5
6  class Libro {
7      public:
8      char nombre[10];
9      Persona *pPersona;

```

```

10
11     Libro();
12     ~Libro();
13 };
14
15 #endif /*LIBRO_H*/
16
17
18 #include <iostream>
19 #include "Persona.h"
20 #include "Libro.h"
21
22 Libro::Libro() {
23     nombre[0]='\0';
24     pPersona=NULL;
25 }
26
27 Libro::~~Libro() {
28     if(pPersona!=NULL)
29         for(int i=0; i<5; i++)
30             if (pPersona->pLibrosPres[i]==this)
31                 pPersona->pLibrosPres[i]=NULL;
32 }
33

```

Clase Persona

```

1  #ifndef PERSONA_H_
2  #define PERSONA_H_
3
4  class Libro;
5
6  class Persona {
7      public:
8          Libro *pLibrosPres[5];
9
10         Persona();
11         ~Persona();
12 };
13
14 #endif /*PERSONA_H_*/
15

```

```
16  #include <iostream>
17  #include "Libro.h"
18  #include "Persona.h"
19
20  Persona::Persona() {
21      int i;
22
23      for(i=0; i<5; i++)
24          pLibrosPres[i]=NULL;
25  }
26
27  Persona::~~Persona() {
28      int i;
29      for(i=0; i<5; i++)
30          if(pLibrosPres[i]!=NULL)
31              pLibrosPres[i]->pPersona=NULL;
32  }
```

19.5 Asociaciones en Java

Como en Java el manejo de objetos es mediante referencias, la implementación de la asociación se simplifica en la medida que la sintaxis de Java es más simple.

Ejemplo: un código que guarda una asociación bidireccional entre clases A y B.

```
1 class A{
2     //lista de atributos
3
4     B      pB;
5 }
6
7 class B{
8     //lista de atributos
9     A      pA;
10 }
```

En el ejemplo anterior se presenta una relación bidireccional, por lo que cada clase tiene su respectiva referencia a la clase contraria de la relación. Además, deben proporcionarse métodos de acceso a la clase relacionada por medio de la referencia.

Una asociación unidireccional del ejemplo anterior sería más simple. Veamos el código si se requiere únicamente una relación de A a B.

Ejemplo:

```
1 class A{
2     //lista de atributos
3     B      pB;
4 }
5
6 class B{
7     //lista de atributos
8 }
```

Recordar

Si el objeto **A** crea al objeto **B**, es responsabilidad de **A** eliminar a la instancia **B** antes de que **A** sea eliminada. En caso contrario, si **B** es independiente de la instancia **A**, **A** debería enviar un mensaje al objeto **B** para que asigne *null* al apuntador de **B** o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida.

En Java, ya que cuenta con un recolector de basura, la importancia radicaría en asegurarnos de no mantener enlaces a objetos que ya no son necesarios.

19.5.1 Multiplicidad de una asociación en Java

La forma de implementar en Java este tipo de relaciones puede variar, pero la más común es por medio de referencias a objetos. Suponiendo que tenemos relaciones bidireccionales:

- **"uno a uno"**. Una referencia de cada lado de la relación, como se ha visto anteriormente.
- **"uno a muchos"**. Una referencia de un lado y un arreglo de referencias a objetos del otro lado.

```
1 class A{  
2     ...  
3     B pB;  
4 }  
5 class B{  
6     A p[];  
7 }
```

Al igual que en C++, es posible manejar una clase independiente que agrupe a pares de direcciones en un objeto independiente de la clase². Por ejemplo, en una estructura de lista.

- **"muchos a muchos"**. Normalmente se utiliza un objeto u objetos independientes que mantiene las relaciones entre los objetos, de manera similar a la solución descrita en el punto anterior.

Ejemplo: Se muestra un código simplificado para manejo de asociaciones.

²Ver figura en tema correspondiente de C++

```
1  //clase Libro
2  class Libro {
3      private String nombreLibro;
4      public Alumno pAlumno;
5
6      public Libro(){
7          //al momento de crearse la instancia, no existe
8          // relación con ningún Alumno
9          pAlumno=null;
10     }
11
12     protected void finalize(){
13         //si es diferente de null, el libro está
14         //asignado a algún Alumno
15         if (pAlumno!=null)
16             //busca la referencia de Alumno a
17             //Libro para ponerla en null
18             for(int i=0; i<5; i++)
19                 if (pAlumno.pLibrosPres[i]==this)
20                     pAlumno.pLibrosPres[i]=null;
21     }
22 }
23
24 //clase Alumno
25 class Alumno {
26     public Libro pLibrosPres[];
27     public Alumno(){
28         int i;
29         //se asume una multiplicidad de 5
30         pLibrosPres = new Libro[5]
31         for(i=0; i<5; i++)
32             pLibrosPres[i]=null;
33     }
34
35     protected void finalize(){
36
37         //pone en null todas las asociaciones de los Libros
38         // a su instancia de Alumno que se elimina
39         for(int i=0; i<5; i++)
40             if(pLibrosPres[i]!=null)
41                 pLibrosPres[i].pAlumno=null;
```

```
42     }  
43 }
```

Este es un ejemplo parcial de cómo se soluciona el manejo de asociaciones entre clases, ya que además se deben de agregar métodos para establecer y eliminar la asociación, en ambas clases si es una asociación bidireccional, o en una clase únicamente si se trata de una asociación unidireccional. Esos deben de ser los únicos métodos que tengan el control sobre los atributos que mantienen la asociación y no deberían ser manejados directamente, por lo que no deben ser públicos como aquí se presentaron.

Una definición más completa - sin implementación - de la clase *Libro* se aprecia a continuación:

```
1  class Libro {  
2      private String nombreLibro;  
3      private String clave;  
4      public Alumno pAlumno;  
5  
6      public Libro() {  
7  
8      public Libro( Alumno pAlumno) {  
9  
10     public String getNombreLibro() {  
11  
12     public void setNombreLibro(String n) {  
13  
14     public String getClave() {  
15  
16     public void setClave(String cve) {  
17  
18     public boolean setAsociacion(Alumno pAlumno) {  
19  
20     public boolean unsetAsociacion() {  
21  
22     public Alumno getAlumno() {  
23  
24     protected finalize {}  
25 }
```

Este sería un estilo más apropiado para el desarrollo de asociaciones, aunque existen otros más elaborados.

19.6 Asociaciones entre Clases en lenguajes dinámicos como Python o Ruby

El manejo de asociaciones en Python o Ruby puede hacerse de forma similar a como se mencionó en Java. De hecho es todavía más simple la definición debido a que no es necesaria la declaración de tipos.

Ejemplo: un código en Ruby que guarda una asociación bidireccional entre clases A y B.

```
1 class A
2     #lista de atributos
3     @pB
4 end
5
6 class B
7     #lista de atributos
8     @pA
9 end
```

Veamos el código si se requiere únicamente una relación de A a B:

```
1 class A
2     #lista de atributos
3     @pB
4 end
5
6 class B
7     #lista de atributos
8 end
```

19.6.1 Multiplicidad de una asociación

La forma de implementar en lenguajes dinámicos este tipo de relaciones puede variar, pero la más común es por medio de referencias a objetos, de forma similar a la usada en Java.

Actividad

Ejercicio 1:

Programa una lista ligada de enteros orientada a objetos. Desarrollarla con una clase *Lista* que contenga al menos los métodos:

- `insertaInicio(Nodo)`
- `insertaFinal(Nodo)`
- `eliminaInicio()`
- `eliminaFinal()`
- `recorreLista()`

Lista está asociada con la clase *Nodo* en una asociación (de nombre *primerNodo*) unidireccional hacia *Nodo* con multiplicidad de 0 a 1. La clase *Nodo* contiene un atributo:

- `dato` - de tipo entero

y los métodos:

- `getDato()`
- `setDato(dato)`
- `getSiguiente()`
- `setSiguiente(Nodo sig)`

El *Nodo* mantiene una asociación reflexiva unidireccional con una multiplicidad de 0 a 1.

Ejercicio 2: Crear una clase *Alumno* que contiene un atributo *matrícula*, *grupo* y un objeto *nombre*. Además mantiene una asociación bidireccional de "uno a muchos" con objetos de una clase *Libro*. La clase *Libro* contiene el *nombre del libro*, *edición*, *año* y tiene un arreglo de 3 objetos de la clase *Nombre* para identificar al autor.

El método *prestamo()* establecería la asociación y el método *devolución()* eliminaría la asociación entre un alumnos y un libro.

El código de prueba debe presentar un menú que permita establecer y deshacer asociaciones entre alumnos y libros. Un alumno puede tener hasta 3 libros prestados al mismo tiempo.

Ejercicio 3: Modificar el ejercicio anterior y en lugar de que *grupo* sea un atributo simple, deberá ser una clase *Grupo* que contenga *carrera*, el *semestre* y que mantenga una relación con un máximo de 30 alumnos. Cuando se elimine al último alumno, el grupo debe desaparecer.

Capítulo 20

Objetos compuestos

Algunas veces una clase no puede modelar adecuadamente una entidad basándose únicamente en tipos de datos simples. Los LPOO permiten a una clase **contener** objetos. Un objeto forma parte directamente de la clase en la que se encuentra declarado.

El objeto compuesto es una especie de relación, pero con una asociación más fuerte con los objetos relacionados. A la noción de objeto **compuesto** se le conoce también como objeto **complejo** o **agregado**.

Concepto
Rumbaugh define a la agregación como <i>"una forma fuerte de asociación, en la cual el objeto agregado está formado por componentes. Los componentes forman parte del agregado. El agregado, es un objeto extendido que se trata como una unidad en muchas operaciones, aún cuando conste físicamente de varios objetos menores."</i> [?]

Ejemplo: Un automóvil se puede considerar ensamblado o agregado, donde el motor y la carrocería serían sus componentes.

El concepto de agregación puede ser relativo a la conceptualización que se tenga de los objetos que se quieren modelar.

Dicho concepto implica obviamente cierta dependencia entre los objetos, por lo que hay que tener en cuenta que pasa con los objetos que son parte del objeto compuesto cuando éste último se destruye. En general tenemos dos opciones:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.
2. Cuando el agregado es destruido también sus componentes se destruyen.

Un objeto que es parte de otro objeto, puede a su vez ser un objeto compuesto. De esta forma podemos tener múltiples niveles. Un objeto puede ser un **agregado recursivo**, es decir, tener un objeto de su misma clase.

Ejemplo: Directorio de archivos.

Sin embargo, la forma en que se implemente la agregación puede no permitir la agregación recursiva.

20.1 Objetos compuestos en C++

Por el momento vamos a considerar la segunda opción mencionada anteriormente, por ser más fácil de implementar y porque es la acción natural de los objetos que se encuentran embebidos como un atributo más en una clase.

Ejemplo:

```

1  class Nombre {
2      private:
3          char paterno[20],
4          materno[20],
5          nom[15];
6      public:
7          set(char *, char*, char *);
8          ...
9  };
10
11 class Persona {
12     private:
13     int edad;
14     Nombre nombrePersona;
15     ...
16 };

```

Al crear un objeto compuesto, cada uno de sus componentes es creado con sus respectivos constructores. Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos *set* correspondientes a la modificación de los atributos de los objetos componentes.
2. Pasar en el constructor del objeto compuesto los argumentos a los constructores de los objetos componentes.

Sintaxis

```

<clase>::<constructor>(<lista de argumentos>)
    : <objeto componente 1>(<lista de argumentos sin el tipo>),...

```

donde la lista de argumentos del objeto compuesto debe incluir a los argumentos de los objetos componentes, para que puedan ser pasados en la creación del objeto.

Tema sugerido

Apéndice X: UMLGEC++

Ejemplo:

```

1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4
5  class Nombre {
6      char *nombre, *paterno, *materno;
7      public:
8          Nombre(char *n, char *p, char*m) {
9              nombre=new char[strlen(n)+1];
10             paterno=new char[strlen(p)+1];
11             materno=new char[strlen(m)+1];
12             strcpy(nombre, n);
13             strcpy(paterno, p);
14             strcpy(materno, m);
15         }
16         ~Nombre() {
17             cout<<"destructor de Nombre: "<<nombre<<endl;
18             delete []nombre;
19             delete []paterno;
20             delete []materno;
21         }
22     };
23
24     class Persona{
25         Nombre miNombre;
26         int edad;
27         public:
28         Persona(char *n, char *p, char*m): miNombre(n, p, m){
29             edad=0;
30         }
31     };

```

```
32
33 int main() {
34     Persona *per1;
35     per1= new Persona("uno", "dos", "tres");
36     Persona per2("Bob", "the", "builder");
37     delete per1;
38     return 0;
39 }
```

Listing 184. Ejemplo de composición en C++.

20.2 Objetos compuestos en Java

En Java, puede no existir mucha diferencia entre la implementación de la asociación y la agregación, debido a que en Java los objetos siempre son manejados por referencias, pero el concepto se debe tener en cuenta para su manejo, además de ser relevante a nivel de diseño de software.

Recordemos que en general hay dos opciones para el manejo de la agregación:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.
2. Cuando el agregado es destruido también sus componentes se destruyen.

Al igual que en C++, vamos a considerar la segunda opción, por ser más fácil de implementar y es la acción natural de los objetos que se encuentran embebidos como un atributo más una clase.

Ejemplo:

```
1  class Nombre {
2      private String paterno;
3      private String materno;
4      private String nom;
5
6      public      set(String pat, String mat, String n) {
7          ...
8      }
9          ...
10 }
11
12 class Persona {
13     private int edad;
14     private Nombre nombrePersona;
15     ...
16 }
```

A diferencia de lo que sucede en C++, los atributos compuestos no tienen memoria asignada, es decir, los objetos compuestos no han sido realmente creados en el momento en que se crea el objeto componente. Es responsabilidad del constructor del objeto componente inicializar los objetos miembros o compuestos, si es que así se requiere.

Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos set correspondientes a la modificación de los atributos de los objetos componentes, esto claro está, después de asignarle la memoria a los objetos componentes.
2. Llamar a algún constructor especializado del objeto componente en el momento de crearlo.

Ejemplo:

```
1  //Programa Persona
2  class Nombre {
3      private String nombre,
4                  paterno,
5                  materno;
6      public Nombre(String n, String p, String m){
7          nombre= new String(n);
8          paterno= new String(p);
9          materno= new String(m);
10     }
11 }
12
13 public class Persona{
14     private Nombre miNombre;
15     private int edad;
16     public Persona(String n, String p, String m) {
17         miNombre= new Nombre(n, p, m);
18         edad=0;
19     }
20
21     public static void main(String args[]) {
22         Persona per1;
23         per1= new Persona("uno", "dos", "tres");
24         Persona per2= new Persona("mi nombre", "mi apellido",
25                                   "otro apellido");
26     }
27 }
```

Listing 185. Ejemplo de composición en Java.

Pero también es posible que un objeto sea un agregado recursivo, es decir, tener como parte de su componente un objeto de su misma clase. Considerar por ejemplo un directorio de archivos, donde cada directorio puede contener, además de archivos, a otros directorios¹.

¹Lo importante aquí es considerar en que solo existe la **posibilidad** de contener un objeto de si

20.3 Objetos compuestos en Ruby

En Ruby, como en Java, puede no existir mucha diferencia entre la implementación de la asociación y la agregación, debido a que en Ruby los objetos siempre son manejados por referencias, pero el concepto se debe tener en cuenta para su manejo, además de ser relevante a nivel de diseño de software.

Al igual que en C++ y Java, vamos a considerar la opción en la cual los componentes son destruidos cuando el agregado es destruido.

Ejemplo:

```
1 class Nombre
2     @paterno
3     @materno
4     @nom
5
6     def set (paterno, materno, nombre)
7         ...
8     end
9     ...
10 end
11
12 class Persona
13     @edad
14     @nombrePersona
15     ...
16 end
```

Como ya se sabe, los identificadores para los atributos no tienen un tipo asociado, sino que son asignados a un objeto en tiempo de ejecución. Es responsabilidad del inicializador del objeto componente inicializar los objetos miembros o compuestos, si es que así se requiere.

```
1 #Programa Persona
2 class Nombre
```

mismo. Si esto fuera una condición obligatoria y no opcional, estaríamos definiendo un **objeto infinito**. Este problema se ve reflejado en lenguajes como C++, donde la forma más simple de implementar la agregación es definiendo un objeto al cual se le asigna espacio en tiempo de compilación, generando entonces el problema de que cada objeto debe reservar memoria para sus componentes, por lo que el compilador no permite que de esta manera se autocontenga. En Java esto no generaría problema porque implícitamente todos los atributos que no son datos simples requieren de una asignación de memoria dinámica.


```
3      @nombre
4      @paterno
5      @materno
6      def initialize(nombre, paterno, materno)
7          @nombre= String.new nombre
8          @paterno= String.new paterno
9          @materno= String.new materno
10     end
11 end
12
13 class Persona
14     @miNombre
15     @edad
16     def initialize(n, p, m)
17         @miNombre = Nombre.new(n, p, m)
18         @edad =0
19     end
20 end
21
22 #código de prueba
23 perl= Persona.new("uno", "dos", "tres")
24 per2= Persona.new("mi nombre", "mi apellido", "otro apellido")
```

Listing 186. Ejemplo de composición en Ruby.

```
1  # Clase MyDate
2  class MyDate
3      @month  # 1-12
4      @day    # 1-31 dependiendo del mes
5      @yea;   # cualquier año
6      def initialize (mn, dy, yr)
7          if ( mn > 0 && mn <= 12 )
8              @month = mn;
9          else
10             @month = 1;
11             puts "Mes #{mn} invalido. Se asigno el mes 1."
12         end
13
14         @year = yr;
15         @day = checkDay( dy );           # validar el dia
16     end
```

```

17     puts "Inicializador de objeto MyDate para fecha " + to_s
18     end
19
20     # verifica que el dia sea correcto de acuerdo al mes
21     def checkDay (testDay)
22         daysPerMonth = [0, 31, 28, 31, 30,
23                         31, 30, 31, 31, 30,
24                         31, 30, 31]
25
26         if ( testDay > 0 && testDay <= daysPerMonth[@month] )
27             return testDay
28         end
29
30         if ( @month == 2 && # Febrero, si el año es bisiestro
31             testDay == 29 &&
32             ( @year % 400 == 0 ||
33               ( @year % 4 == 0 && @year % 100 != 0 ) ) )
34             return testDay
35         end
36
37         puts "Dia #{testDay.to_s} invalido. Se asigno el dia 1."
38
39         return 1 # deja al objeto en un estado consistente
40     end
41
42     def to_s
43         return @month.to_s + "/" + @day.to_s + "/" + @year.to_s
44     end
45 end
46
47 # Clase Empleado
48 class Employee
49     @firstName
50     @lastName
51     @birthDate
52     @hireDate
53
54     def initialize (fName, lName, bMonth, bDay, bYear,
55                   hMonth, hDay, hYear)
56         @firstName = fName
57         @lastName = lName
58         @birthDate = MyDate.new( bMonth, bDay, bYear )

```

```
59     @hireDate = MyDate.new( hMonth, hDay, hYear )
60   end
61
62   def to_s
63     return @lastName + ", " + @firstName +
64           "  Contratado: " + @hireDate.to_s +
65           "  Fecha nacimiento: " + @birthDate.to_s
66   end
67 end
68 # clase EmployeeTest
69 class EmployeeTest
70   attr_reader :e
71   @e
72
73   def initialize
74     @e = Employee.new( "Juanito", "Sanchez", 7, 24, 49,
75                       3, 12, 88 )
76   end
77
78 end
79
80 # codigo de prueba
81 et= EmployeeTest.new
82 puts et.e.to_s
```

Listing 187. Ejemplo 2 de composición en Ruby.

Capítulo 21

Polimorfismo

El polimorfismo es la capacidad de ofrecer una interfaz para distintos tipos, de manera que un tipo polimórfico es al que se le pueden aplicar operaciones con distintos tipos. Existen distintos tipos de polimorfismo:

- **Polimorfismo ad-hoc[?]**. Es cuando una función tiene un conjunto de implementaciones distintas sobre un rango de tipos de datos y sus combinaciones. Este tipo de polimorfismo es soportado en muchos lenguajes por medio de la sobrecarga y es también conocido como **polimorfismo estático**.
- **Polimorfismo de subtipo o de inclusión¹[?]**. Es el tipo de polimorfismo más común, en el que un conjunto de instancias de distintas clases están relacionadas por una superclase. Tan común que es lo que muchas veces se explica como polimorfismo. También conocido como **polimorfismo dinámico**.
- **Polimorfismo paramétrico[?]**. Cuando se escribe código sin especificar el tipo que va a ser usado. En POO es conocido como programación genérica. En programación funcional es llamado simplemente polimorfismo.

¹"Polymorphic types are types whose operations are applicable to values of more than one type."

Capítulo 22

Polimorfismo de subtipos

Concepto
"La capacidad de polimorfismo permite crear programas con mayores posibilidades de expansiones futuras, aún para procesar en cierta forma objetos de clases que no han sido creadas o están en desarrollo." [?]

Concepto
El polimorfismo se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método. [?]

Tener en cuenta que no es lo mismo que simplemente sobrecribir un método de clase base en una clase derivada, pues como se vio anteriormente, si se tiene a un apuntador de clase base y a través de el se hace la llamada a un método, se ejecuta el método de la clase base independientemente del objeto referenciado por el apuntador. Este no es un comportamiento polimórfico.

22.1 Polimorfismo y funciones virtuales C++

En C++, el polimorfismo se implementa a través de clases derivadas y **funciones virtuales**. Al hacer una solicitud de un método, a través de un apuntador a clase base para usar un método virtual, C++ determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Concepto

Una función virtual es un método miembro declarado como **virtual** en una clase base y siendo este método sobrescrito en una o más clases derivadas.

Las funciones virtuales son muy especiales, debido a que cuando una función es accedida por un apuntador a una clase base, y éste mantiene una referencia a un objeto de una clase derivada, el programa determina en tiempo de ejecución a que función llamar, de acuerdo al tipo de objeto al que se apunta. Esto se conoce como **ligadura tardía**¹ y el compilador de C++ incluye en el código máquina el manejo de ese tipo de asociación de métodos.

La utilidad se da cuando se tiene un método en una clase base, y éste es declarado virtual. De esta forma, cada clase derivada puede tener su propia implementación del método si es que así lo requiere la clase; y si un apuntador a clase base hace referencia a cualquiera de los objetos de clases derivadas, se determina dinámicamente cual de todos los métodos debe ejecutar.

La sintaxis en C++ implica declarar al método de la clase base con la palabra reservada *virtual*, redefiniendo ese método en cada una de las clases derivadas.

Al declarar un método como virtual, este método se conserva así a través de toda la jerarquía de herencia, del punto en que se declaró hacia abajo. Aunque de este modo no es necesario volver a usar la palabra virtual en ninguno de los métodos inferiores del mismo nombre, es posible declararlo de forma explícita para que el programa sea más claro.

Es importante señalar que las funciones virtuales que sean sobrescritas en clases derivadas, deben tener además de la misma firma que la función virtual base, el mismo tipo de retorno.

¹Término opuesto a **ligadura temprana** o **ligadura estática**, la cual asocia los métodos en tiempo de compilación.

Sintaxis

```
class base {  
    virtual <tipo> <método> (<parámetros>;  
};
```

Ejemplo:

```
1  //ejemplo funciones virtuales  
2  #include <iostream>  
3  using namespace std;  
4  
5  class base {  
6  public:  
7      virtual void quien() {  
8          cout<<"base\n";  
9      }  
10 };  
11  
12 class primera: public base {  
13 public:  
14     void quien() {  
15         cout<<"primera\n";  
16     }  
17 };  
18  
19 class segunda: public base {  
20 public:  
21     void quien() {  
22         cout<<"segunda\n";  
23     }  
24 };  
25  
26 class tercera: public base {  
27 };  
28  
29 class cuarta: public base {  
30 public:  
31     //No se vale con un tipo de dato diferente
```

```
32         /*int quien(){
33             cout<<"cuarta\n";
34             return 1;
35         }*/
36     };
37
38     int main() {
39         base objBase, *pBase;
40         primera obj1;
41         segunda obj2;
42         tercera obj3;
43         cuarta obj4;
44
45         pBase=&objBase;
46         pBase->quien();
47
48         pBase=&obj1;
49         pBase->quien();
50
51         pBase=&obj2;
52         pBase->quien();
53
54         pBase=&obj3;
55         pBase->quien();
56
57         pBase=&obj4;
58         pBase->quien();
59
60         return 0;
61     }
```

Listing 188. Ejemplo de polimorfismo en C++.

Hay que hacer notar que las funciones virtuales pueden seguirse usando sin apunadores, mediante un objeto de la clase. De esta forma, el método a ejecutar se determina de manera estática; es decir, en tiempo de compilación (**ligadura estática**). Obviamente el método a ejecutar es aquel definido en la clase del objeto o el heredado de su clase base, si la clase derivada no lo redefinió.

La sobrecarga no utiliza ligadura dinámica. Esta es resuelta en tiempo de compilación. Si se declara en una clase derivada un método con otro tipo de dato como retorno, el compilador manda un error, ya que esto no es permitido.

Si se declara un método con el mismo nombre pero diferentes parámetros, la función virtual queda desactivada de ese punto hacia abajo en la jerarquía de he-

rencia.

22.1.1 Clase abstracta y clase concreta en C++

Existen clases que son útiles para representar una estructura en particular, pero que no van a tener la necesidad de generar objetos directamente a partir de esa clase, éstas se conocen como **clases abstractas**, o de manera más apropiada como **clases base abstractas**, puesto que sirven para definir una estructura jerárquica.

La clase base abstracta entonces, tiene como objetivo proporcionar una clase base que ayude al modelado de la jerarquía de herencia, aunque esta sea muy general y no sea práctico tener instancias de esa clase.

Por lo tanto, de una clase abstracta no se pueden tener objetos, mientras que en clases a partir de las cuales se puedan instanciar objetos se conocen como **clases concretas**.

En C++, una clase se hace abstracta al declarar **al menos uno** de los métodos virtuales como puro. Un método o función virtual pura es aquel que en su declaración tiene el inicializador de `= 0`.

Sintaxis

```
virtual <tipo> <nombre>(<parámetros>) =0;    //virtual pura
```

Es importante tener en cuenta que una clase sigue siendo abstracta hasta que no se implemente la función virtual pura, en una de las clases derivadas. Si no se hace la implementación, la función se hereda como virtual pura y por lo tanto la clase sigue siendo considerada como abstracta.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener apuntadores a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante los apuntadores a la clase abstracta.

22.1.2 Destructores virtuales

Cuando se aplica la instrucción *delete* a un apuntador de clase base, será ejecutado el destructor de la clase base sobre el objeto, independientemente de la clase a la que pertenezca. La solución es declarar al destructor de la clase base como virtual. De esta forma al borrar a un objeto se ejecutará el destructor de la clase a la que pertenezca el objeto referenciado, a pesar de que los destructores no tengan el mismo nombre.

Un constructor no puede ser declarado como virtual.

Ejemplos de funciones virtuales y polimorfismo:

[Ejemplo: Programa de cálculo de salario.](#)

```
1  // EMPLEADO.H
2  // Abstract base class Employee
3  #ifndef EMPLEADO_H_
4  #define EMPLEADO_H_
5
6  class Employee {
7      public:
8          Employee(const char *, const char *);
9          ~Employee();
10         const char *getFirstName() const;
11         const char *getLastName() const;
12
13         virtual float earnings() const = 0; // virtual pura
14         virtual void print() const = 0;     // virtual pura
15     private:
16         char *firstName;
17         char *lastName;
18 };
19
20 #endif /*EMPLEADO_H_*/
21
22 // EMPLEADO.CPP
23 #include <iostream>
24 #include <string>
25 #include <assert.h>
26 #include "empleado.h"
27
28 Employee::Employee(const char *first, const char *last)
29 {
30     firstName = new char[ strlen(first) + 1 ];
31     assert(firstName != 0);
32     strcpy(firstName, first);
33
34     lastName = new char[ strlen(last) + 1 ];
35     assert(lastName != 0);
36     strcpy(lastName, last);
37 }
38
39 Employee::~~Employee()
40 {
41     delete [] firstName;
```

```
42     delete [] lastName;
43 }
44
45 const char *Employee::getFirstName() const
46 {
47
48     return firstName;
49 }
50
51 const char *Employee::getLastName() const
52 {
53     return lastName;
54 }
55 // JEFE.H
56 // Clase derivada de empleado
57 #ifndef JEFE_H_
58 #define JEFE_H_
59
60 #include "empleado.h"
61
62 class Boss : public Employee {
63     public:
64         Boss(const char *, const char *, float = 0.0);
65         void setWeeklySalary(float);
66         virtual float earnings() const;
67         virtual void print() const;
68     private:
69         float weeklySalary;
70 };
71
72 #endif /*JEFE_H_*/
73
74 // JEFE.CPP
75 #include <iostream>
76 #include "jefe.h"
77 using namespace std;
78 Boss::Boss(const char *first, const char *last, float s)
79     : Employee(first, last)
80 { weeklySalary = s > 0 ? s : 0; }
81
82 void Boss::setWeeklySalary(float s)
83 { weeklySalary = s > 0 ? s : 0; }
```

```

84
85 float Boss::earnings() const { return weeklySalary; }
86
87 void Boss::print() const
88 {
89     cout << "\n           Jefe: " << getFirstName()
90         << " " << getLastName();
91 }
92
93
94 // COMIS.H
95 // Trabajador por comisión derivado de Empleado
96 #ifndef COMIS_H_
97 #define COMIS_H_
98 #include "empleado.h"
99
100 class CommissionWorker : public Employee {
101     public:
102         CommissionWorker(const char *, const char *,
103                         float = 0.0, float = 0.0, int = 0);
104         void setSalary(float);
105         void setCommission(float);
106         void setQuantity(int);
107         virtual float earnings() const;
108         virtual void print() const;
109     private:
110         float salary;      // salario base por semana
111         float commission; // comisión por cada venta
112         int quantity;      // cantidad de elementos vendidos por semana
113 };
114
115 #endif /*COMIS_H_*/
116
117 // COMIS.CPP
118 #include <iostream>
119 #include "comis.h"
120 using namespace std;
121
122 CommissionWorker::CommissionWorker(const char *first,
123                                   const char *last, float s, float c, int q)
124     : Employee(first, last)
125 {

```

```

126     salary = s > 0 ? s : 0;
127     commission = c > 0 ? c : 0;
128     quantity = q > 0 ? q : 0;
129 }
130
131
132 void CommissionWorker::setSalary(float s)
133 { salary = s > 0 ? s : 0; }
134
135 void CommissionWorker::setCommission(float c)
136 { commission = c > 0 ? c : 0; }
137
138 void CommissionWorker::setQuantity(int q)
139 { quantity = q > 0 ? q : 0; }
140
141 float CommissionWorker::earnings() const
142 { return salary + commission * quantity; }
143
144 void CommissionWorker::print() const
145 {
146     cout << "\nTrabajador por comision: " << getFirstName()
147         << ' ' << getLastName();
148 }
149
150 // PIEZA.H
151 // Trabajador por pieza derivado de Empleado
152 #ifndef PIEZA_H_
153 #define PIEZA_H_
154
155 #include "empleado.h"
156
157 class PieceWorker : public Employee {
158     public:
159         PieceWorker(const char *, const char *,
160             float = 0.0, int = 0);
161         void setWage(float);
162         void setQuantity(int);
163         virtual float earnings() const;
164         virtual void print() const;
165     private:
166         float wagePerPiece; // pago por cada pieza
167         int quantity;       // piezas por semana

```

```
168     };
169
170     #endif /*PIEZA_H_*/
171
172     // PIEZA.CPP
173     #include <iostream>
174     #include "pieza.h"
175     using namespace std;
176
177     // Constructor for class PieceWorker
178     PieceWorker::PieceWorker(const char *first,
179                             const char *last, float w, int q)
180         : Employee(first, last)
181     {
182         wagePerPiece = w > 0 ? w : 0;
183         quantity = q > 0 ? q : 0;
184     }
185
186     void PieceWorker::setWage(float w)
187     { wagePerPiece = w > 0 ? w : 0; }
188
189     void PieceWorker::setQuantity(int q)
190     { quantity = q > 0 ? q : 0; }
191
192     float PieceWorker::earnings() const
193     { return quantity * wagePerPiece; }
194
195     void PieceWorker::print() const {
196         cout << "\n    Tabajador por pieza: " << getFirstName()
197              << ' ' << getLastName();
198     }
199
200     // HORA.H
201     // Trabajador por hora derivado de Empleado
202     #ifndef HORA_H_
203     #define HORA_H_
204
205     #include "empleado.h"
206
207     class HourlyWorker : public Employee {
208     public:
209         HourlyWorker(const char *, const char *,
```



```

210         float = 0.0, float = 0.0);
211     void setWage(float);
212     void setHours(float);
213     virtual float earnings() const;
214     virtual void print() const;
215 private:
216     float wage;    // salario por hora
217     float hours;   // horas trabajadas en la semana
218 };
219
220 #endif /*HORA_H*/
221
222 // HORA.CPP
223 #include <iostream>
224 #include "hora.h"
225 using namespace std;
226
227 HourlyWorker::HourlyWorker(const char *first, const char *last,
228                             float w, float h)
229     : Employee(first, last)
230 {
231     wage = w > 0 ? w : 0;
232     hours = h >= 0 && h < 168 ? h : 0;
233 }
234
235 void HourlyWorker::setWage(float w) { wage = w > 0 ? w : 0; }
236
237 void HourlyWorker::setHours(float h)
238 { hours = h >= 0 && h < 168 ? h : 0; }
239
240 float HourlyWorker::earnings() const { return wage * hours; }
241
242 void HourlyWorker::print() const
243 {
244     cout << "\n    Trabajador por hora: " << getFirstName()
245          << ' ' << getLastName();
246 }
247
248 // main.cpp
249 #include <iostream>
250 #include <iomanip>
251 #include "empleado.h"

```

```
252 #include "jefe.h"
253 #include "comis.h"
254 #include "pieza.h"
255 #include "hora.h"
256 using namespace std;
257
258 int main() {
259     // formato de salida
260     cout << setprecision(2);
261
262     Employee *ptr; // apuntador a clase base
263
264     Boss b("John", "Smith", 800.00);
265     ptr = &b; // apuntador de clase base apuntando a objeto de clase derivada
266     ptr->print(); // ligado dinámico
267     cout << " ganado $" << ptr->earnings(); // ligado dinámico
268     b.print(); // ligado estático
269     cout << " ganado $" << b.earnings(); // ligado estático
270
271     CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
272     ptr = &c;
273     ptr->print();
274     cout << " ganado $" << ptr->earnings();
275     c.print();
276     cout << " ganado $" << c.earnings();
277
278     PieceWorker p("Bob", "Lewis", 2.5, 200);
279     ptr = &p;
280     ptr->print();
281     cout << " ganado $" << ptr->earnings();
282     p.print();
283     cout << " ganado $" << p.earnings();
284
285     HourlyWorker h("Karen", "Precio", 13.75, 40);
286     ptr = &h;
287     ptr->print();
288     cout << " ganado $" << ptr->earnings();
289     h.print();
290     cout << " ganado $" << h.earnings();
291
292     cout << endl;
293     return 0;
```

294

}

Listing 189. Ejemplo de funciones virtuales y polimorfismo en C++. Programa de cálculo de salario..

Ejemplo: Programa de figuras geométricas con una interfaz abstracta Shape (Forma)

```

1  // Figura.H
2  #ifndef figura_H
3  #define figura_H
4
5  class Shape {
6  public:
7      virtual float area() const { return 0.0; }
8      virtual float volume() const { return 0.0; }
9      virtual void printShapeName() const = 0; // virtual pura
10 };
11 #endif
12
13 // Punto.H
14 #ifndef PUNTO_H_
15 #define PUNTO_H_
16 #include <iostream>
17 #include "figura.h"
18
19 class Point : public Shape {
20     friend ostream &operator<<(ostream &, const Point &);
21 public:
22     Point(float = 0, float = 0);
23     void setPoint(float, float);
24     float getX() const { return x; }
25     float getY() const { return y; }
26     virtual void printShapeName() const { cout << "Punto: "; }
27 private:
28     float x, y;
29 };
30 #endif /*PUNTO_H_*/
31
32 // Punto.CPP
33 #include <iostream.h>
34 #include "punto.h"
35 Point::Point(float a, float b)

```

```
36 {
37     x = a;
38     y = b;
39 }
40
41 void Point::setPoint(float a, float b)
42 {
43     x = a;
44     y = b;
45 }
46
47 ostream &operator<<(ostream &output, const Point &p)
48 {
49     output << '[' << p.x << ", " << p.y << ']';
50     return output;
51 }
52
53 // Circulo.H
54 #ifndef CIRCULO_H_
55 #define CIRCULO_H_
56 #include "punto.h"
57
58 class Circle : public Point {
59     friend ostream &operator<<(ostream &, const Circle &);
60 public:
61     Circle(float r = 0.0, float x = 0.0, float y = 0.0);
62
63     void setRadius(float);
64     float getRadius() const;
65     virtual float area() const;
66     virtual void printShapeName() const { cout << "Circulo: "; }
67 private:
68     float radius;
69 };
70 #endif /*CIRCULO_H_*/
71
72 // Circulo.CPP
73 #include <iostream>
74 #include <iomanip>
75 #include "circulo.h"
76 using namespace std;
77
```

```

78 Circle::Circle(float r, float a, float b)
79     : Point(a, b)
80     { radius = r > 0 ? r : 0; }
81
82 void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }
83
84 float Circle::getRadius() const { return radius; }
85
86 float Circle::area() const { return 3.14159 * radius * radius; }
87
88 ostream &operator<<(ostream &output, const Circle &c)
89 {
90     output << '[' << c.getX() << ", " << c.getY()
91         << "]; Radio=" << setprecision(2) << c.radius;
92
93     return output;
94 }
95
96 // Cilindro.H
97 #ifndef CILINDRO_H_
98 #define CILINDRO_H_
99 #include "circulo.h"
100
101 class Cylinder : public Circle {
102     friend ostream &operator<<(ostream &, const Cylinder &);
103 public:
104     Cylinder(float h = 0.0, float r = 0.0,
105             float x = 0.0, float y = 0.0);
106
107     void setHeight(float);
108     virtual float area() const;
109     virtual float volume() const;
110     virtual void printShapeName() const { cout << "Cilindro: "; }
111 private:
112     float height; // altura del cilindro
113 };
114 #endif /*CILINDRO_H_*/
115
116 // Cilindro.CPP
117 #include <iostream>
118 #include <iomanip>
119 #include "cilindro.h"

```

```
120
121 Cylinder::Cylinder(float h, float r, float x, float y)
122     : Circle(r, x, y)
123 { height = h > 0 ? h : 0; }
124
125 void Cylinder::setHeight(float h)
126     { height = h > 0 ? h : 0; }
127
128 float Cylinder::area() const
129 {
130     return 2 * Circle::area() +
131         2 * 3.14159 * Circle::getRadius() * height;
132 }
133
134 float Cylinder::volume() const
135 {
136     float r = Circle::getRadius();
137     return 3.14159 * r * r * height;
138 }
139
140 ostream &operator<<(ostream &output, const Cylinder& c)
141 {
142     output << '[' << c.getX() << ", " << c.getY()
143         << "]; Radio=" << setprecision(2) << c.getRadius()
144         << "; Altura=" << c.height;
145     return output;
146 }
147
148 // main.CPP
149 #include <iostream>
150 #include <iomanip>
151 using namespace std;
152
153 #include "figura.h"
154 #include "punto.h"
155 #include "circulo.h"
156 #include "cilindro.h"
157
158 int main() {
159     Point point(7, 11);
160     Circle circle(3.5, 22, 8);
161     Cylinder cylinder(10, 3.3, 10, 10);
```

```
162
163     point.printShapeName();           // ligado estático
164     cout << point << endl;
165
166     circle.printShapeName();
167     cout << circle << endl;
168
169     cylinder.printShapeName();
170     cout << cylinder << "\n\n";
171     cout << setprecision(2);
172     Shape *ptr;                       // apuntador de clase base
173
174     // apuntador de clase base referenciando objeto de clase derivada
175     ptr = &point;
176     ptr->printShapeName();             // ligado dinámico
177     cout << "x = " << point.getX() << "; y = " << point.getY()
178         << "\nArea = " << ptr->area()
179         << "\nVolumen = " << ptr->volume() << "\n\n";
180
181     ptr = &circle;
182     ptr->printShapeName();
183     cout << "x = " << circle.getX() << "; y = " << circle.getY()
184         << "\nArea = " << ptr->area()
185         << "\nVolumen = " << ptr->volume() << "\n\n";
186
187     ptr = &cylinder;
188     ptr->printShapeName();             // dynamic binding
189     cout << "x = " << cylinder.getX() << "; y = " << cylinder.getY()
190         << "\nArea = " << ptr->area()
191         << "\nVolumen = " << ptr->volume() << endl;
192     return 0;
193 }
```

Listing 190. Ejemplo de polimorfismo en C++. Programa de figuras geométricas con una interfaz abstracta Shape (Forma).

22.2 Polimorfismo y clases abstractas Java

El polimorfismo es implementado en Java a través de clases derivadas y clases abstractas.

Concepto

Recordar: El **polimorfismo** se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método.

Al hacer una solicitud de un método, a través de una variable de referencia a clase base para usar un método, Java determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Los métodos en Java - a diferencia de C++ - tienen este comportamiento por default, debido a que cuando un método es accedido por una referencia a una clase base, y esta mantiene una referencia a un objeto de una clase derivada, el programa determina **en tiempo de ejecución** a que método llamar, de acuerdo al tipo de objeto al que se apunta.

Esto como ya se ha visto, se conoce como **ligadura tardía** y permite otro nivel de reutilización de código, resaltado por la simplificación con respecto a C++ de no tener que declarar al método como virtual.

Ejemplo:

```
1 //ejemplo Prueba
2 class base {
3     public void quien() {
4         System.out.println("base");
5     }
6 }
7
8 class primera extends base {
9     public void quien() {
10        System.out.println("primera");
11    }
12 }
13
14 class segunda extends base {
15     public void quien() {
16        System.out.println("segunda");
17    }
```



```
18 }
19
20 class tercera extends base { }
21
22 class cuarta extends base {
23     /*public int quien(){    No se vale con un tipo de dato diferente
24         System.out.println("cuarta");
25         return 1;
26     }*/
27 }
28
29 public class Prueba {
30     public static void main(String args[]) {
31         base objBase= new base(), pBase;
32         primera obj1= new primera();
33         segunda obj2= new segunda();
34         tercera obj3= new tercera();
35         cuarta obj4= new cuarta();
36
37         pBase=objBase;
38         pBase.quien();
39
40         pBase=obj1;
41         pBase.quien();
42
43         pBase=obj2;
44         pBase.quien();
45
46         pBase=obj3;
47         pBase.quien();
48
49         pBase=obj4;
50         pBase.quien();
51     }
52 }
```

Listing 191. Ejemplo de polimorfismo en Java.

Como se aprecia en el ejemplo anterior, en caso de que el método no sea sobrescrito, se ejecuta el método de la clase base.

Es importante señalar que – al igual que en C++– los métodos que sean sobrescritos en clases derivadas, deben tener además de la misma firma que método base, el mismo tipo de retorno. Si se declara en una clase derivada un método con

otro tipo de dato como retorno, se generará un error en tiempo de compilación.

22.2.1 Clase abstracta y clase concreta en Java

Concepto
Recordar: Una clase base abstracta , es aquella que es definida para especificar características generales que van a ser aprovechadas por sus clases derivadas, pero no se necesita instanciar a dicha superclase.
Sintaxis
<pre> abstract class ClaseAbstracta { //código de la clase } </pre>

Además, existe la posibilidad de contar con métodos abstractos:

Concepto
Un método abstracto lleva la palabra reservada <i>abstract</i> y contiene sólo el nombre y su firma. No necesita implementarse, ya que esto es tarea de las subclases.

Si una clase contiene al menos un método abstracto, toda la clase es considerada abstracta y es conveniente, por claridad, declararla como tal. Es posible claro, declarar a una clase como abstracta sin que tenga métodos abstractos.

Ejemplo básico para un método abstracto:

```

1 abstract class ClaseAbstracta {
2
3     public abstract void noTengoCodigo( int x);
4
5 }

```

Si se crea una subclase de una clase que contiene un método abstracto, deberá de especificarse el código de ese método; de lo contrario, el método seguirá siendo abstracto y por consecuencia también lo será la subclase².

²En C++, una clase se hace abstracta al declarar al menos uno de los métodos virtuales como puro.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener referencias a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante las referencias a la clase abstracta.

El uso de clases abstractas **fortalece** al polimorfismo, al poder partir de clases definidas en lo general, sin implementación de código, pero pudiendo ser agrupadas todas mediante variables de referencia a las clases base.

Ejemplos de clases abstractas y polimorfismo:

Programa de cálculo de salario

```
1  // Clase base abstracta Employee
2  public abstract class Employee {
3      private String firstName;
4      private String lastName;
5
6      // Constructor
7      public Employee( String first, String last ) {
8          firstName = new String ( first );
9          lastName = new String( last );
10     }
11
12     public String getFirstName() {
13         return new String( firstName );
14     }
15
16     public String getLastName() {
17         return new String( lastName );
18     }
19
20     // el metodo abstracto debe de ser implementado por cada
21     // clase derivada de Employee para poder ser
22     // instanciadas las subclases
23     public abstract double earnings();
24 }
25
26 // Clase Boss class derivada de Employee
27 public final class Boss extends Employee {
28     private double weeklySalary;
29
30     public Boss( String first, String last, double s ) {
31         super( first, last ); // llamada al constructor de clase base
32         setWeeklySalary( s );
33     }
```

```
34
35     public void setWeeklySalary( double s ){
36         weeklySalary = ( s > 0 ? s : 0 );
37     }
38
39     // obtiene pago del jefe
40     public double earnings() {
41         return weeklySalary;
42     }
43
44     public String toString() {
45         return "Jefe: " + getFirstName() + ' ' +
46             getLastName();
47     }
48 }
49
50 // Clase PieceWorker derivada de Employee
51 public final class PieceWorker extends Employee {
52     private double wagePerPiece; // pago por pieza
53     private int quantity;        // piezas por semana
54
55     public PieceWorker( String first, String last,
56         double w, int q )    {
57         super( first, last );
58         setWage( w );
59         setQuantity( q );
60     }
61
62     public void setWage( double w )    {
63         wagePerPiece = ( w > 0 ? w : 0 );
64     }
65
66     public void setQuantity( int q )    {
67         quantity = ( q > 0 ? q : 0 );
68     }
69
70     public double earnings()    {
71         return quantity * wagePerPiece;
72     }
73
74     public String toString()    {
75         return "Trabajador por pieza: " +
```

[illegible]

```
118         super( first, last );
119         setSalary( s );
120         setCommission( c );
121         setQuantity( q );
122     }
123
124     public void setSalary( double s )    {
125         salary = ( s > 0 ? s : 0 );
126     }
127
128     public void setCommission( double c )    {
129         commission = ( c > 0 ? c : 0 );
130     }
131
132     public void setQuantity( int q )    {
133         quantity = ( q > 0 ? q : 0 );
134     }
135
136     public double earnings()    {
137         return salary + commission * quantity;
138     }
139
140     public String toString()    {
141         return "Trabajador por Comision : " +
142             getFirstName() + ' ' + getLastName();
143     }
144 }
145
146 // Programa de ejemplo Polimorfismo
147 public class Polimorfismo {
148     public static void main( String rgs[] ) {
149         Employee ref; // referencia de clase base
150         Boss b;
151         CommissionWorker c;
152         PieceWorker p;
153         HourlyWorker h;
154         b = new Boss( "Alan", "Turing", 800.00 );
155         c = new CommissionWorker( "Ada", "Lovelace",
156                                 400.0, 3.0, 150 );
157         p = new PieceWorker( "Grace", "Hopper", 2.5, 200 );
158         h = new HourlyWorker( "James", "Gosling", 13.75, 40 );
159     }
```

```

160         ref = b; // referencia de superclase a objeto de subclase
161         System.out.println( ref.toString() + " ganó $" +
162             ref.earnings() );
163         System.out.println( b.toString() + " ganó $" +
164             b.earnings() );
165
166         ref = c; // referencia de superclase a objeto de subclase
167         System.out.println( ref.toString() + " ganó $" +
168             ref.earnings() );
169         System.out.println( c.toString() + " ganó $" +
170             c.earnings() );
171
172         ref = p; // referencia de superclase a objeto de subclase
173         System.out.println( ref.toString() + " ganó $" +
174             ref.earnings() );
175         System.out.println( p.toString() + " ganó $" +
176             p.earnings() );
177
178         ref = h; // referencia de superclase a objeto de subclase
179         System.out.println( ref.toString() + " ganó $" +
180             ref.earnings() );
181         System.out.println( h.toString() + " ganó $" +
182             h.earnings() );
183     }
184 }

```

Listing 192. Ejemplo de clase abstracta y polimorfismo en Java. Programa de cálculo de salario

Ejemplo: Programa de figuras geométricas con una clase abstracta Shape (Forma)

```

1 // Definicion de clase base abstracta Shape
2 public abstract class Shape {
3
4     public double area() {
5         return 0.0;
6     }
7
8     public double volume() {
9         return 0.0;
10    }
11

```

```
12     public abstract String getName();
13 }
14
15 // Definicion de clase Point
16 public class Point extends Shape {
17     protected double x, y; // coordenadas del punto
18
19     public Point( double a, double b ) { setPoint( a, b ); }
20
21     public void setPoint( double a, double b )    {
22         x = a;
23         y = b;
24     }
25
26     public double getX() { return x; }
27
28     public double getY() { return y; }
29
30     public String toString()
31     { return "[" + x + ", " + y + "]; }
32
33     public String getName() {
34         return "Punto";
35     }
36 }
37
38 // Definicion de clase Circle
39 public class Circle extends Point { // hereda de Point
40     protected double radius;
41
42     public Circle()    {
43         super( 0, 0 );
44         setRadius( 0 );
45     }
46
47     public Circle( double r, double a, double b )    {
48         super( a, b );
49         setRadius( r );
50     }
51
52     public void setRadius( double r )
53     { radius = ( r >= 0 ? r : 0 ); }
```



```
54     public double getRadius() { return radius; }
55
56
57     public double area() { return 3.14159 * radius * radius; }
58
59     public String toString()
60 { return "Centro = " + super.toString() +
61     "; Radio = " + radius; }
62
63     public String getName() {
64         return "Circulo";
65     }
66 }
67
68 // Definicion de clase Cylinder
69 public class Cylinder extends Circle {
70     protected double height; // altura del cilindro
71
72     public Cylinder( double h, double r, double a, double b )
73     {
74         super( r, a, b );
75         setHeight( h );
76     }
77
78     public void setHeight( double h ){
79         height = ( h >= 0 ? h : 0 );
80     }
81
82     public double getHeight() {
83         return height;
84     }
85
86     public double area() {
87         return 2 * super.area() +
88             2 * 3.14159 * radius * height;
89     }
90
91     public double volume() {
92         return super.area() * height;
93     }
94
95     public String toString(){
96         return super.toString() + "; Altura = " + height;
```

```
96     }
97
98     public String getName() {
99         return "Cilindro";
100     }
101 }
102
103 // Codigo de prueba
104 public class Polimorfismo02 {
105
106     public static void main (String args []) {
107         Point point;
108         Circle circle;
109         Cylinder cylinder;
110         Shape arrayOfShapes[];
111
112         point = new Point( 7, 11 );
113         circle = new Circle( 3.5, 22, 8 );
114         cylinder = new Cylinder( 10, 3.3, 10, 10 );
115
116         arrayOfShapes = new Shape[ 3 ];
117
118         // asigno las referencias de los objetos de subclase
119         // a un arreglo de superclase
120         arrayOfShapes[ 0 ] = point;
121         arrayOfShapes[ 1 ] = circle;
122         arrayOfShapes[ 2 ] = cylinder;
123
124         System.out.println( point.getName() + ": " + point.toString());
125
126         System.out.println( circle.getName() + ": " + circle.toString());
127
128         System.out.println( cylinder.getName() + ": " + cylinder.toString());
129
130         for ( int i = 0; i < 3; i++ ) {
131             System.out.println( arrayOfShapes[ i ].getName() +
132                                 ": " + arrayOfShapes[ i ].toString());
133             System.out.println( "Area = " + arrayOfShapes[ i ].area() );
134             System.out.println( "Volume = " + arrayOfShapes[ i ].volume() );
135         }
136     }
```

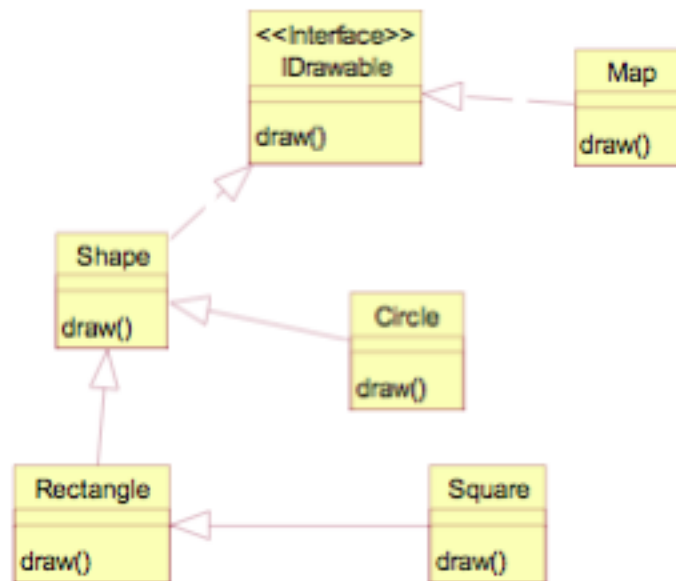


Fig. 22.1. Ejemplo de polimorfismo con interfaces en Java

137

}

Listing 193. Ejemplo de polimorfismo en Java. Programa de figuras geométricas con una clase abstracta Shape (Forma)

Clase abstracta o interfaz

22.2.2 Ejemplo de Polimorfismo con una Interfaz en Java

Los programas anteriores estaban basados en clases y clases abstractas. Sin embargo, también es posible tener variables de referencia a interfaces, a través de las cuales se implemente el polimorfismo. El siguiente programa muestra otra estructura clásica de clases “gráficas”, todas contienen su propia implementación de *draw()*, y son organizadas en dos arreglos de ejemplo: uno de la clase principal, y el segundo del tipo de la interfaz.

Ejemplo:

```

1  //programa Polimorfismo
2  interface IDrawable {
3      void draw();
4  }
5
6  class Shape implements IDrawable {

```

```
7     public void draw() { System.out.println("Dibujando Figura."); }
8 }
9
10 class Circle extends Shape {
11     public void draw() { System.out.println("Dibujando Circulo."); }
12 }
13
14 class Rectangle extends Shape {
15     public void draw() { System.out.println("Dibujando Rectangulo."); }
16 }
17
18 class Square extends Rectangle {
19     public void draw() { System.out.println("Dibujando cuadrado."); }
20 }
21
22 class Map implements IDrawable {
23     public void draw() { System.out.println("Dibujando mapa."); }
24 }
25
26 public class Polimorfismo03 {
27     public static void main(String args[]) {
28         Shape[] shapes = {new Circle(), new Rectangle(), new Square()};
29         IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};
30
31         System.out.println("Dibujando figuras:");
32         for (int i = 0; i < shapes.length; i++)
33             shapes[i].draw();
34
35         System.out.println("Dibujando elementos dibujables:");
36         for (int i = 0; i < drawables.length; i++)
37             drawables[i].draw();
38     }
39 }
```

Listing 194. Ejemplo de polimorfismo con interfaces en Java.

22.3 Polimorfismo en Ruby

El polimorfismo es implementado en Ruby a través de clases derivadas.

Al hacer una solicitud de un método, a través de una variable, Ruby determina **en tiempo de ejecución** a que método llamar, de acuerdo al tipo de objeto al que se apunta.

```
1  #ejemplo Prueba
2  class Base
3      def quien
4          puts "base"
5      end
6  end
7
8  class Primera < Base
9      def quien
10         puts "primera"
11     end
12 end
13
14 class Segunda < Base
15     def quien
16         puts "segunda"
17     end
18 end
19
20 class Tercera < Base
21
22 end
23
24 class Cuarta < Base
25     def quien  #Si se vale con un tipo de retorno diferente (definido dinamicament
26         puts "cuarta"
27         return 1
28     end
29 end
30
31 #codigo de prueba
32 def prueba
33     objBase= Base.new
34     obj1= Primera.new
35     obj2= Segunda.new
```

```
36     obj3= Tercera.new
37     obj4= Cuarta.new
38
39     pBase=objBase
40     pBase.quien
41
42     pBase=obj1
43     pBase.quien
44
45     pBase=obj2
46     pBase.quien
47
48     pBase=obj3
49     pBase.quien
50
51     pBase=obj4
52     pBase.quien
53 end
54
55 prueba
```

Listing 195. Ejemplo de polimorfismo en Ruby.

Como se aprecia en el ejemplo anterior, en caso de que el método no sea sobrescrito, se ejecuta el método de la clase base.

Es importante señalar que **no** es necesario que el tipo de retorno coincida, pues éste se determina dinámicamente.

22.3.1 ¿Y la clase abstracta?

Ruby no tiene el concepto de clase abstracta y clase concreta. Si por cuestión de diseño es necesario definir una clase que no debe permitir instancias, es posible crear un método en dicha clase como sigue:

```
1 class MiClase
2     def self.abstract?
3         return self == MiClase
4     end
5 end
```

Esto no limita realmente a que alguien pueda instanciar la clase, pero es posible preguntar si la clase es “abstract?”.

Programa de cálculo de salario

```
1  # Clase base Employee
2  class Employee
3      attr_reader :firstName, :lastName
4
5      @firstName
6      @lastName
7
8      def self.abstract?
9          return self == Employee
10     end
11
12     # Inicializador
13     def initialize (first, last)
14         @firstName = String.new( first )
15         @lastName = String.new( last )
16     end
17
18     # método sin código
19     def earnings
20     end
21 end
22
23 # Clase Boss cderivada de Employee
24 class Boss < Employee
25     @weeklySalary
26
27     def initialize (first, last, s)
28         super( first, last ) # llamada al constructor de clase base
29         setWeeklySalary( s )
30     end
31
32     def setWeeklySalary( s )
33         @weeklySalary = ( s > 0 ? s : 0 )
34     end
35
36     # obtiene pago del jefe
37     def earnings
38         return @weeklySalary
39     end
40
```

```
41         def to_s
42             return "Jefe: " + @firstName + " " + @lastName
43         end
44     end
45
46     # Clase PieceWorker derivada de Employee
47     class PieceWorker < Employee
48         attr_reader :wagePerPiece, :quantity
49
50         @wagePerPiece # pago por pieza
51         @quantity # piezas por semana
52
53         def initialize(first, last, w, q )
54             super( first, last )
55             setWage( w )
56             setQuantity( q )
57         end
58
59         def setWage( w )
60             @wagePerPiece = ( w > 0 ? w : 0 )
61         end
62
63         def setQuantity( q )
64             @quantity = ( q > 0 ? q : 0 )
65         end
66
67         def earnings
68             return quantity * wagePerPiece
69         end
70
71         def to_s
72             return "Trabajador por pieza: " + @firstName + " " + @lastName
73         end
74     end
75
76     # Clase HourlyWorker derivada de Employee
77     class HourlyWorker < Employee
78         attr_reader :wage, :hours
79
80         @wage # pago por hora
81         @hours # horas trabajadas por semana
82     end
```



```
83     def initialize(first, last, w, h)
84         super( first, last )
85         setWage( w )
86         setHours( h )
87     end
88
89     def setWage( w )
90         @wage = ( w > 0 ? w : 0 )
91     end
92
93     def setHours( h )
94         @hours = ( h >= 0 && h < 168 ? h : 0 )
95     end
96
97     def earnings
98         return @wage * @hours
99     end
100
101     def to_s
102         return "Trabajador por hora: " + @firstName + " " + @lastName
103     end
104 end
105
106 # Clase CommissionWorker derivada de Employee
107 class CommissionWorker < Employee
108     attr_reader :salary, :commission, :quantity
109
110     @salary      # salario base por semana
111     @commission  # monto por producto vendido
112     @quantity    # total de productos vendidos por semana
113
114     def initialize(first, last, s, c, q)
115         super( first, last )
116         setSalary( s )
117         setCommission( c )
118         setQuantity( q )
119     end
120
121     def setSalary( s )
122         @salary = ( s > 0 ? s : 0 )
123     end
124
```

```
125     def setCommission( c )
126         @commission = ( c > 0 ? c : 0 )
127     end
128
129     def setQuantity( q )
130         @quantity = ( q > 0 ? q : 0 )
131     end
132
133     def earnings
134         return @salary + @commission * @quantity
135     end
136
137     def to_s
138         return "Trabajador por Comision : " + @firstName + " " + @lastName
139     end
140 end
141
142
143 # Codigo de ejemplo Polimorfismo
144 b = Boss.new( "Alan", "Turin", 800.00 )
145 c = CommissionWorker.new( "Ada", "Lovelace", 400.0, 3.0, 150 )
146 p = PieceWorker.new( "Grace", "Hopper", 2.5, 200 )
147 h = HourlyWorker.new( "James", "Gosling", 13.75, 40 )
148
149 puts Employee.abstract?
150
151 ref = b # referencia a objeto de subclase
152 puts ref.to_s + " gana $" + ref.earnings.to_s
153 puts b.to_s + " gana $" + b.earnings.to_s
154
155 ref = c # referencia a objeto de subclase
156 puts ref.to_s + " gana $" + ref.earnings.to_s
157 puts c.to_s + " gana $" + c.earnings.to_s
158
159 ref = p # referencia a objeto de subclase
160 puts ref.to_s + " gana $" + ref.earnings.to_s
161 puts p.to_s + " gana $" + p.earnings.to_s
162
163 ref = h # referencia a objeto de subclase
164 puts ref.to_s + " gana $" + ref.earnings.to_s
165 puts h.to_s + " gana $" + h.earnings.to_s
```

Listing 196. Ejemplo polimorfismo en Ruby. Programa de cálculo de salario.
Programa de figuras geométricas con una clase abstracta Shape (Forma)

```
1  # Definicion de clase base abstracta Shape
2  class Shape
3      def self.abstract?
4          return self == Shape
5      end
6
7      def area
8          return 0.0
9      end
10
11     def volume
12         return 0.0
13     end
14
15     def getName
16     end
17 end
18
19 # Definicion de clase Point
20 class Point < Shape
21     attr_reader :x, :y
22
23     @x
24     @y # coordenadas del punto
25
26     def initialize (a, b)
27         setPoint( a, b )
28     end
29
30     def setPoint(a, b)
31         @x, @y = a, b
32     end
33
34     def to_s
35     return "[" + @x.to_s + ", " + @y.to_s + "]"
36     end
37
38     def getName
39         return "Punto"
```

```
40         end
41     end
42
43     # Definicion de clase Circle
44     class Circle < Point # hereda de Point
45         attr_reader :radius
46
47         @radius
48
49         def initialize(r, a, b)
50             super( a, b )
51             setRadius( r )
52         end
53
54         def setRadius(r)
55             @radius = ( r >= 0 ? r : 0 )
56         end
57
58         def area
59             return 3.14159 * radius * radius
60         end
61
62         def to_s
63             return "Centro = " + super + "; Radio = " + @radius.to_s
64         end
65
66         def getName
67             return "Circulo "
68         end
69     end
70
71     # Definicion de clase Cylinder
72     class Cylinder < Circle
73         attr_reader :height
74         alias_method :areaCircle, :area #define un nuevo nombre para el metodo area de
75         # o puede usarse alias :areaCircle :area
76
77         @height # altura del cilindro
78
79         def initialize(h, r, a, b)
80             super(r, a, b)
81             setHeight(h)
```

```
82         end
83
84         def setHeight(h)
85             @height = ( h >= 0 ? h : 0 )
86         end
87
88         def area
89             return 2 * super +          2 * 3.14159 * @radius * @height
90         end
91
92         def volume
93             return areaCircle * @height
94         end
95
96         def to_s
97             return super + "; Altura = " + @height.to_s
98         end
99
100        def getName
101            return "Cilindro "
102        end
103    end
104
105    # Codigo de prueba
106    #sh = Shape()           # genera error
107    point = Point.new( 7, 11 )
108    circle = Circle.new( 3.5, 22, 8 )
109    cylinder = Cylinder.new( 10, 3.3, 10, 10 )
110
111    # asigno las referencias de los objetos de subclase
112    # a un arreglo
113    arrayOfShapes=[]
114    arrayOfShapes[0 ] = point
115    arrayOfShapes[1 ] = circle
116    arrayOfShapes[2 ] = cylinder
117
118    puts point.getName + ": " + point.to_s
119
120    puts circle.getName + ": " + circle.to_s
121
122    dos?puts cylinder.getName + ": " + cylinder.to_s
123
```

```
124 for elem in arrayOfShapes
125     puts elem.getName + elem.to_s
126     puts "  Area = " + elem.area.to_s
127     puts "  Volumen = " + elem.volume.to_s
128 end
```

Listing 197. Ejemplo polimorfismo en Ruby. Programa de figuras geométricas con una clase abstracta Shape (Forma).

22.4 Polimorfismo en Python

Debido a que Python es un lenguaje tipado dinámicamente, el polimorfismo se ejecuta en automático. De hecho, cada operación es una operación polimórfica en Python. Cualquier método aplicado a un objeto funcionará mientras la clase del objeto la soporte, siendo determinado en tiempo de ejecución si es posible llevarse a cabo la operación.

Ejemplo:

```
1  # Ejemplo de polimorfismo en Python
2  class base:
3      def quien(self):
4          print('base')
5  class primera (base):
6      def quien(self):
7          print('primera')
8  class segunda (base):
9      def quien(self):
10         print('segunda')
11 class tercera (base):
12     def foo(self):
13         return
14 #script de ejecución
15 objBase = base()
16 obj1 = primera()
17 obj2 = segunda()
18 obj3 = tercera()
19
20 pBase= objBase
21 pBase.quien()
22 pBase= obj1
23 pBase.quien()
24 pBase= obj2
25 pBase.quien()
26 pBase= obj3
27 pBase.quien()
```

Listing 198. Ejemplo de polimorfismo en Python.

22.4.1 Clases abstractas y polimorfismo en Python

El concepto de clases abstractas no está implementado directamente en Python. Sin embargo, si proporciona un módulo para clases bases abstractas (ABC – Abstract Base Class)³. El módulo proporciona una metaclasses⁴ ABCMeta y una clase de ayuda ABC⁵. Una clase que tiene una metaclasses derivada de ABCMeta no puede ser instanciada. Se pueden definir métodos abstractos mediante el uso del decorador `@abstractmethod`. El uso de este decorador requiere que la metaclasses de la clase sea ABCMeta o se derive de esta.

Sintaxis

```
from abc import ABC

class <MiClaseAbstracta> (ABC) :
    <resto de la clase>
```

Ejemplo:

```
1  #polimorfismo y clase abstracta en Python
2  from abc import ABC, abstractmethod
3
4  class Shape(ABC):
5      def area(self):
6          return 0
7      def volume(self):
8          return 0
9
10     @abstractmethod
11     def getName(self):
12         pass
13
14     class Point (Shape):
15         def __init__(self, a, b):
16             self.x = a
```

³[abc](#)

⁴[Metaclasses](#)

⁵El concepto de metaclasses en Python va más allá del alcance del curso.


```
17         self.y = b
18
19     def setPoint(self, a, b):
20         self.x = a
21         self.y = b
22
23     def getX(self):
24         return self.x
25
26     def getY(self):
27         return self.y
28
29     def toString(self):
30         return "[" + str(self.x) + ", " + str(self.y) + "]"
31
32     def getName(self):
33         return 'Punto'
34
35 class Circle (Point):
36     def __init__(self, r, a, b):
37         super().__init__(a, b)
38         self.radius = r
39
40     def setRadius(self, r):
41         if r>=0 :
42             self.radius = r
43         else:
44             self.radius = 0
45
46     def area(self):
47         return 3.14159 * self.radius * self.radius
48
49     def toString(self):
50         return "Centro = " + super().toString() + "; Radio = " + str(self.radius)
51
52     def getName(self):
53         return 'Círculo'
54
55 class Cylinder (Circle):
56     def __init__(self, h, r, a, b):
57         super().__init__(r, a, b)
58         self.height = h
```

```
59
60     def setHeight(self, h):
61         if h>=0:
62             self.height = h
63         else:
64             self.height = 0
65
66     def getHeight(self):
67         return self.height
68
69     def area(self):
70         return 2 * super().area() + 2 * 3.14159 * self.radius * self.height
71
72     def volume(self):
73         return super().area() * self.height
74
75     def toString(self):
76         return super().toString() + "; Altura = " + str(self.height)
77
78     def getName(self):
79         return 'Cilindro'
80
81     #script de prueba
82     point = Point(7, 11)
83     circle = Circle(3.5, 22, 8)
84     cylinder = Cylinder(10, 3.3, 10, 10)
85     arrayOfShapes = [point, circle, cylinder]
86
87     print(point.getName() + ": " + point.toString())
88     print(circle.getName() + ": " + circle.toString())
89     print(cylinder.getName() + ": " + cylinder.toString())
90
91     for sh in arrayOfShapes:
92         print(sh.getName() + ": " + sh.toString())
93         print( "Area = " + str(sh.area()) )
94         print( "Volume = " + str(sh.volume()) )
```

Listing 199. Ejemplo de clases abstractas y polimorfismo en Python.

Capítulo 23

Polimorfismo paramétrico: programación genérica

La programación genérica favorece la reutilización de código, permitiendo que se generen objetos específicos para un tipo a partir de **clases genéricas**. Las clases genéricas son conocidas también como **plantillas de clase** o **clases parametrizadas**.

23.1 Plantillas de clase en C++

Antes se mencionó el uso de plantillas en C++ aplicado a funciones. El concepto de plantillas es aplicable también a la programación orientada a objetos en C++ a través de **plantillas de clase**.

El uso de plantillas de clase no es diferente al uso de plantillas en operaciones no orientadas a objetos:

Sintaxis
<pre>template <class T> ○ template <typename T></pre>

Veamos el ejemplos clásicos aprovechando el uso de plantillas.

[Ejemplo:](#)

```
1  #include <iostream>
2
3  using namespace std;
4
5  template <class T, class U>
6  class Pair {
7  private:
8      T first;
9      U second;
10 public:
11     Pair(T f, U second) {
12         first=f;
13         this->second=second;
14
15     }
16
17     T getFirst() {
18         return first;
19 }
```

```

20     }
21     U getSecond() {
22         return second;
23     }
24 };
25
26 int main() {
27     Pair<float, int> *pair = new Pair<float, int>(10.5,2);
28
29     cout<<"Obten el primer elemento:" << pair->getFirst()<<endl;
30     cout<<"Obten el segundo elemento:"<< pair->getSecond()<<endl;
31     return 0;
32 }

```

Listing 200. Ejemplo de plantillas en C++.

Ejemplo:

```

1  // stack.h
2  // Clase de plantilla Pila
3  #ifndef STACK_H_
4  #define STACK_H_
5
6  template< class T >
7  class Stack {
8  public:
9      Stack( int = 10 );
10     ~Stack() { delete [] stackPtr; }
11     char push( const T& );
12     char pop( T& );
13 private:
14     int size;
15     int top;
16     T *stackPtr;
17
18     char isEmpty() const { return top == -1; }
19     char isFull() const { return top == size - 1; }
20 };
21
22
23 template< class T >
24 Stack< T >::Stack( int s )

```

```
25 {
26     size = s > 0 ? s : 10;
27     top = -1;
28     stackPtr = new T[ size ];
29 }
30
31 template< class T >
32 char Stack< T >::push( const T &pushValue )
33 {
34     if ( !isFull() ) {
35         stackPtr[ ++top ] = pushValue;
36         return 1;
37     }
38     return 0;
39 }
40
41 template< class T >
42 char Stack< T >::pop( T &popValue )
43 {
44     if ( !isEmpty() ) {
45         popValue = stackPtr[ top-- ];
46         return 1;
47     }
48     return 0;
49 }
50
51 #endif /*STACK_H_*/
52
53 // Ejemplo uso de plantillas de clase
54 #include <iostream>
55 #include "stack.h"
56
57 using namespace std;
58
59 int main() {
60     Stack< double > doubleStack( 5 );
61     double f = 1.1;
62     cout << "Insertando elementos en doubleStack \n";
63
64     while ( doubleStack.push( f ) ) {
65         cout << f << ' ';
66         f += 1.1;
```

```

67     }
68
69     cout << "\nLa pila está llena. No se puede insertar el elemento " << f
70         << "\n\nSacando elementos de doubleStack\n";
71
72     while ( doubleStack.pop( f ) )
73         cout << f << ' ';
74
75     cout << "\nLa pila está vacía. No se pueden eliminar más elementos\n";
76
77     Stack< int > intStack;
78     int i = 1;
79     cout << "\nInsertando elementos en intStack\n";
80
81     while ( intStack.push( i ) ) {
82         cout << i << ' ';
83         ++i;
84     }
85
86     cout << "\nLa pila está llena. " << i
87         << "\n\nSacando elementos de intStack\n";
88
89     while ( intStack.pop( i ) )
90         cout << i << ' ';
91
92     cout << "\nLa pila está vacía. No se pueden eliminar más elementos \n";
93     return 0;
94 }

```

Listing 201. Ejemplo, una pila con plantillas en C++.

Las plantillas de clase ayudan a la reutilización de código, al permitir varias versiones de clases para un tipo de dato a partir de clases genéricas. A estas clases específicas se les conoce como **clases de plantilla**.

Concepto

Una clase de plantilla es entonces como una instancia de una plantilla de clase.

Con respecto a la herencia en combinación con el uso de plantillas, se deben tener en cuenta las siguientes situaciones[?]:

- Una plantilla de clase se puede derivar de una clase de plantilla.
- Una plantilla de clase se puede derivar de una clase que no sea plantilla.
- Una clase de plantilla se puede derivar de una plantilla de clase.
- Una clase que no sea de plantilla se puede derivar de una plantilla de clase.

En cuanto a los miembros estáticos, cada clase de plantilla que se crea a partir de una plantilla de clases mantiene sus propias copias de los miembros estáticos.

23.2 Standard Template Library (STL)

Las plantillas de clase son una herramienta muy poderosa en C++. Esto ha llevado a desarrollar lo que se conoce como STL. STL es el acrónimo de *Standard Template Library*, y es una biblioteca de C++ que proporciona un conjunto de clases contenedoras, iteradores y de algoritmos genéricos:

- Las clases contenedoras incluyen vectores, listas, deque, conjuntos, multi-conjuntos, multimapas, pilas, colas y colas de prioridad.
- Los iteradores son generalizaciones de apuntadores: son objetos que apuntan a otros objetos. Son usados normalmente para iterar sobre un conjunto de objetos. Los iteradores son importantes porque son típicamente usados como interfaces entre las clases contenedoras y los algoritmos.
- Los algoritmos genéricos incluyen un amplio rango de algoritmos fundamentales para los más comunes tipos de manipulación de datos, como ordenamiento, búsqueda, copiado y transformación.
- STL es una biblioteca estándar de ANSI/ISO desde julio de 1994.

La STL está altamente parametrizada, por lo que casi cada componente en la STL es una plantilla[?]. Podemos usar por ejemplo la plantilla *vector* < *T* > para hacer uso de vectores sin necesidad de preocuparnos del manejo de memoria:

```
1     vector<int> v(3);           // Declara un vector de 3 elementos.
2         v[0] = 7;
3         v[1] = v[0] + 3;
4         v[2] = v[0] + v[1];     // v[0] == 7, v[1] == 10, v[2] == 17
```

Los algoritmos proporcionados por la STL ayudan a manipular los datos de los contenedores[?]. Por ejemplo, podemos invertir el orden de los elementos de un vector, usando el algoritmo *reverse()*:


```
reverse(v.begin(), v.end()); // v[0]==17, v[1]==10, v[2]==7
```

Ejemplo:

```

1  #ifndef STACK_HPP_
2  #define STACK_HPP_
3  #include <vector>
4
5  template <typename T>
6  class Stack {
7      private:
8          std::vector<T> elems;      // elementos
9
10     public:
11         void push(T const&);
12         void pop();
13         T top() const;              // regresa elemento en el tope
14         bool empty() const {        // regresa si la pila esta vacia
15             return elems.empty();
16         }
17     };
18
19     template <typename T>
20     void Stack<T>::push (T const& elem)
21     {
22         elems.push_back(elem);      // añade una copia de elem
23     }
24
25     template<typename T>
26     void Stack<T>::pop ()
27     {
28         if (elems.empty()) {
29             std::cout<<"Stack<>::pop(): pila vacia";
30             return;
31         }
32         elems.pop_back();            // remueve el ultimo elemento
33     }
34
35     template <typename T>
36     T Stack<T>::top () const
37     {
38         if (elems.empty()) {
39             std::cout<<"Stack<>::top(): pila vacia";

```

```
40     }
41     return elems.back();      // regresa copia del elemento en el tope
42 }
43 #endif /*STACK_HPP_*/
44
45 #include <iostream>
46 #include <string>
47 #include <cstdlib>
48 #include "stack.hpp"
49
50 int main()
51 {
52     Stack<int>      intStack;      // pila de enteros
53     Stack<std::string> stringStack; // pila de strings
54
55     // manipulapila de enteros
56     intStack.push(7);
57     std::cout << intStack.top() << std::endl;
58
59     // manipula pila de strings
60     stringStack.push("hola");
61     std::cout << stringStack.top() << std::endl;
62     stringStack.pop();
63     stringStack.pop();
64 }
```

Listing 202. Ejemplo de STL.

Tabla de Clases más Usadas de la STL en C++

Clase	Descripción	Ejemplo
<code>vector</code>	Contenedor dinámico que permite almacenar una colección de elementos en un arreglo redimensionable.	<code>std::vector<int>v = {1, 2, 3}; v.push_back(4);</code>
<code>list</code>	Lista doblemente enlazada que permite inserciones y eliminaciones rápidas en cualquier posición.	<code>std::list<int>l = {1, 2, 3}; l.push_back(4);</code>
<code>deque</code>	Contenedor que permite inserciones y eliminaciones rápidas tanto al principio como al final.	<code>std::deque<int>d; d.push_front(1); d.push_back(2);</code>
<code>set</code>	Contenedor que almacena elementos únicos y los mantiene ordenados automáticamente.	<code>std::set<int>s = {3, 1, 2}; s.insert(4);</code>
<code>map</code>	Contenedor que almacena pares clave-valor, ordenado por las claves.	<code>std::map<std::string, int>m; m[.Alicia"] = 25;</code>
<code>unordered_map</code>	Contenedor similar a <code>map</code> , pero no mantiene los elementos ordenados.	<code>std::unordered_map<std::string, int>um; um[.Alicia"] = 25;</code>
<code>stack</code>	Contenedor que sigue la estructura LIFO (Last In, First Out).	<code>std::stack<int>st; st.push(1); st.push(2);</code>
<code>queue</code>	Contenedor que sigue la estructura FIFO (First In, First Out).	<code>std::queue<int>q; q.push(1); q.push(2);</code>
<code>priority_queue</code>	Cola de prioridad que permite almacenar elementos ordenados según un criterio.	<code>std::priority_queue<int>pq; pq.push(3); pq.push(1);</code>
<code>pair</code>	Estructura que permite almacenar un par de valores relacionados.	<code>std::pair<int, std::string>p = {1, .Alicia"};</code>
<code>tuple</code>	Similar a <code>pair</code> , pero permite almacenar más de dos valores de distintos tipos.	<code>std::tuple<int, std::string, float>t = {1, .Alicia", 3.14};</code>

23.3 Clases Genéricas en Java

Java 1.5 introdujo finalmente el uso de clases genéricas (*generics*)[?]. El uso de clases genéricas es una característica poderosa usada en otros lenguajes, siendo C++ el ejemplo más conocido que soporta programación genérica mediante el uso de plantillas o *templates*.

Sintaxis

```
class NombreClase <Lista de parámetros de tipos> { ... }
```

Ejemplo:

```
1 class Pair<T, U> {
2     private final T first;
3     private final U second;
4     public Pair(T first, U second) { this.first=first; this.second=second; }
5     public T getFirst() { return first; }
6     public U getSecond() { return second; }
7 }
8
9 public class PairExample {
10     public static void main(String[] args) {
11
12         Pair<String, Integer> pair = new Pair<String, Integer>("one",2);
13
14         // no acepta tipos de datos básicos o primitivos
15         //Pair<String, int> pair2 = new Pair<String, Integer>("one",2);
16
17         // siguiente linea generaría un warning de seguridad de tipos
18         //Pair<String, Integer> pair3 = new Pair("one",2);
19
20         System.out.println("Obtén primer elemento:" + pair.getFirst());
21         System.out.println("Obtén segundo elemento:" + pair.getSecond());
22     }
23 }
```

Listing 203. Ejemplo de clases genéricas en Java.

Es también posible parametrizar interfaces, como se muestra a continuación.

Sintaxis

```
interface NombreInterfaz <Lista de parámetros de tipos> { ... }
```

Ejemplo:

```
1 interface IPair<T, U>{
2     public T getFirst();
3     public U getSecond();
4 }
5
6 class Pair<T, U> implements IPair<T, U>{
7     private final T first;
8     private final U second;
9     public Pair(T first, U second) { this.first=first; this.second=second; }
10    public T getFirst() { return first; }
11    public U getSecond() { return second; }
12 }
13
14 public class PairExample {
15     public static void main(String[] args) {
16
17         IPair<String, Integer> ipair = new Pair<String, Integer>("one",2);
18
19         System.out.println("Obtén primer elemento:"+ipair.getFirst());
20         System.out.println("Obtén segundo elemento:"+ipair.getSecond());
21     }
22 }
23 }
```

Listing 204. Ejemplo clases e interfaces genéricas en Java.

Un requerimiento para el uso tipos genéricos en Java es que no pueden usarse tipos de datos primitivos, porque los tipos primitivos o básicos no son subclases de `Object`[?]. Por lo que sería ilegal por ejemplo querer instanciar `Pair < int, String >`. La ventaja es que el uso de la clase `Object` significa que solo un archivo de clase (`.class`) necesita ser generado por cada clase genérica[?].

Restricciones de las clases genéricas, ver¹.

23.3.1 Autoboxing y Unboxing

Aunque los tipos genéricos en Java no aceptan tipos primitivos directamente, Java proporciona un mecanismo llamado **autoboxing** (auto-empaquetado) que convierte automáticamente entre tipos primitivos y sus correspondientes clases wrapper (envolventes). Cada tipo primitivo tiene una clase wrapper correspondiente:

¹Java generics

Tipo Primitivo	Clase Wrapper
int	Integer
double	Double
boolean	Boolean
char	Character
byte	Byte
short	Short
long	Long
float	Float

El **autoboxing** es la conversión automática que hace el compilador de Java de un tipo primitivo a su clase wrapper correspondiente. El **unboxing** es la conversión inversa, de la clase wrapper al tipo primitivo.

Ejemplo:

```
1 List<Integer> numeros = new ArrayList<Integer>();
2
3 // Autoboxing: el int 5 se convierte automáticamente a Integer
4 numeros.add(5); // equivalente a: numeros.add(Integer.valueOf(5));
5
6 // Unboxing: el Integer se convierte automáticamente a int
7 int valor = numeros.get(0); // equivalente a: int valor = numeros.get(0).intValue();
```

Importante

Gracias al autoboxing, es posible usar tipos primitivos con colecciones genéricas de manera transparente. Sin embargo, es importante recordar que internamente se están usando objetos wrapper, lo cual puede tener implicaciones de rendimiento en código crítico debido a la creación de objetos adicionales.

23.4 Biblioteca de Clases Genéricas en Java

Al igual que C++ con la STL, Java tiene un conjunto de clases genéricas predefinidas. Su uso, de manera similar que con las clases genéricas definidas por el programador, no está permitido para tipos primitivos, por lo que solo objetos podrán ser contenidos. Las principales clases genéricas en Java son, como en la STL, clases contenedoras o colecciones². El *Java Collections Framework* (JCF) es un conjunto de interfaces y clases definidos en los paquetes *java.util* y *java.util.concurrent*.

²Las colecciones en Java eran implementadas antes de la versión 1.5 pero sin el uso de clases genéricas. El uso de versiones anteriores de colecciones con colecciones genéricas es permitido por

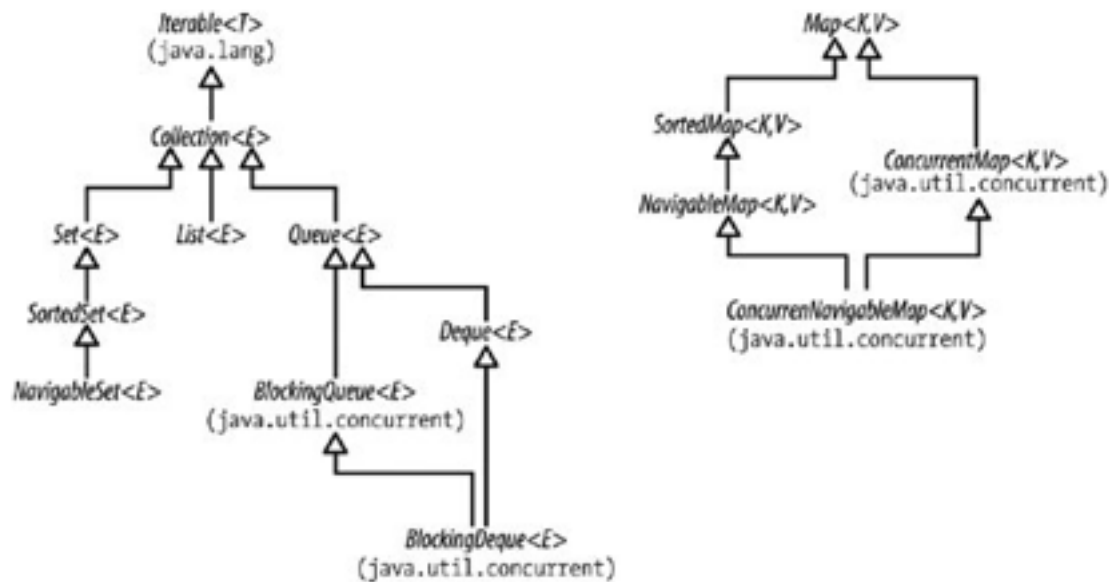


Fig. 23.1. Biblioteca genérica en Java

Las interfaces del JCF son:

- *Collection*. Contiene la funcionalidad básica requerida en casi cualquier colección de objetos (con excepción de *Map*)
- *Set*. Es una colección sin duplicados, donde el orden es no significativo. Sin embargo contiene un método que devuelve el conjunto ordenado (*SortedSet*).
- *Queue*. Define el comportamiento básico de una estructura de cola.
- *List*. Es una colección donde el orden es significativo, permitiendo además valores duplicados.
- *Map*. Define una colección donde un valor clave es asociado para almacenar y recuperar elementos.

La siguiente figura muestra las principales interfaces de la JCF[?]:

Los iteradores son objetos que te permiten recorrer una colección de objetos, obteniendo o removiendo elementos. Un objeto iterador implementa la interfaz *Iterator* o la interfaz *ListIterator*. En general, para usar un iterador para recorrer una colección se debe: 1. Obtener un iterador al inicio de la colección llamando al método *iterator()* de la colección. 2. Definir un ciclo que haga la llamada a *hasNext()*.

compatibilidad hacia atrás pero debe tenerse especial cuidado pues hay situaciones que el compilador no puede validar.

El ciclo iterará mientras el método sea verdadero. 3. En el ciclo, obtener cada elemento llamando al método *next()*.

Ejemplo:

```
1  // Usando la interfaz Collection
2  import java.util.List;
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Iterator;
6
7  public class CollectionTest {
8      private static final String[] colors =
9      { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
10     private static final String[] removeColors =
11     { "RED", "WHITE", "BLUE" };
12
13     // crea ArrayList, añade Colors y la manipula
14     public CollectionTest() {
15         List< String > list = new ArrayList< String >();
16         List< String > removeList = new ArrayList<String>();
17
18         // añade elementos del arreglo colors a list
19         for ( String color : colors )
20             list.add( color );
21
22         // añade elementos del arreglo removeColors a removeList
23         for ( String color : removeColors )
24             removeList.add( color );
25
26         System.out.println( "ArrayList: " );
27         // despliega contenido de list
28         for ( int count = 0; count < list.size(); count++ )
29             System.out.printf( "%s ", list.get( count ) );
30
31         // remueve de list colores contenidos en removeList
32         removeColors( list, removeList );
33
34         System.out.println( "\n\nArrayList después de llamar removeColors: " );
35         // despliega contenido de list
36         for ( String color : list )
37             System.out.printf( "%s ", color );
38     } // end CollectionTest constructor
```



```

39
40     // remueve colores especificados en collection2 de collection1
41     private void removeColors (
42         Collection< String > collection1, Collection< String > collection2 ) {
43         // obtiene iterator
44         Iterator< String > iterator = collection1.iterator();
45
46         // mientras colección tiene elementos
47         while ( iterator.hasNext() )
48             if ( collection2.contains( iterator.next() ) )
49                 iterator.remove(); // remueve color actual
50     }
51
52     public static void main( String args[] ) {
53         new CollectionTest();
54     }
55 }

```

Listing 205. Ejemplo usando la JCF en Java.

23.4.1 Ejemplos complementarios de Clases Genéricas en Java

[Ejemplo](#) de una pila genérica simple³:

```

1  import java.util.*;
2
3  public class PilaGenérica <T> {
4      private ArrayList<T> pila = new ArrayList<T> ();
5      private int tope = 0;
6
7      public int size () {
8          return tope;
9      }
10
11     public void push (T elemento) {
12         pila.add (tope++, elemento);
13     }
14
15     public T pop () {
16         return pila.remove (--tope);

```

³Basado en: [Generic Stack](#)

```
17     }
18
19     public static void main (String[] args) {
20         PilaGenérica<Integer> p = new PilaGenérica<Integer> ();
21
22         p.push (17);
23         int i = p.pop ();
24         System.out.format ("%4d%n", i);
25     }
26 }
```

Listing 206. Ejemplo de una pila genérica simple en Java.

Ejemplo de una pila genérica como lista ligada⁴:

```
1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3
4 public class Pila<Elemento> implements Iterable<Elemento> {
5     private int tamaño;           // tamaño de la pila
6     private Nodo primer;         // tope de la pila
7
8     // clase anidada Nodo
9     private class Nodo {
10         private Elemento elemento;
11         private Nodo siguiente;
12     }
13
14     // Crea una pila vacía.
15     public Pila() {
16         primer = null;
17         tamaño = 0;
18     }
19
20     // Esta vacía la pila?
21     public boolean estaVacía() {
22         return primer == null;
23     }
24
25     // Regresa el número de elementos en la pila
26     public int getTamaño() {
```

⁴Ejemplo basado de: [Stack](#)

```
27     return tamaño;
28 }
29
30 // Añade elemento a la pila.
31 public void push(Elemento elemento) {
32     Nodo viejoPrimer = primer;
33
34     primer = new Nodo();
35     primer.elemento = elemento;
36     primer.siguiente = viejoPrimer;
37     tamaño++;
38 }
39
40 /**
41  *Regresa el elemento en el tope de la pila y lo elimina.
42  *Lanza una excepción si no hay elemento porque la pila este vacía.*//
43 public Elemento pop() {
44     if (estaVacia())
45         throw new RuntimeException("Pila vacía");
46     Elemento elemento = primer.elemento; // guarda elemento para retornarlo
47     primer = primer.siguiente; // elimina el primer nodo
48     tamaño--;
49     return elemento; // regresa elemento
50 }
51
52 /** Regresa el elemento en el tope de la pila sin modificarla.
53  * Lanza una excepción si la pila esta vacía.*//
54 public Elemento ver() {
55     if (estaVacia())
56         throw new RuntimeException("Pila vacía");
57     return primer.elemento;
58 }
59
60 /** Regresa representación en cadena.*//
61 public String toString() {
62     StringBuilder s = new StringBuilder();
63     for (Elemento elemento : this)
64         s.append(elemento + " ");
65     return s.toString();
66 }
67 // Regresa un iterador a la pila que itera a través de los elementos en orden LIFO
68 public Iterator<Elemento> iterator() {
```

```
69         return new ListIterator();
70     }
71     // Iterador, no se implementa remove() dado que es opcional
72     private class ListIterator implements Iterator<Elemento> {
73         private Nodo actual = primer;
74
75         public boolean hasNext() {
76             return actual != null;
77         }
78
79         public void remove() {
80             throw new UnsupportedOperationException();
81         }
82
83         public Elemento next() {
84             if (!hasNext())
85                 throw new NoSuchElementException();
86             Elemento elemento = actual.elemento;
87             actual = actual.siguiente;
88             return elemento;
89         }
90     }
91
92     public static void main(String[] args) {
93         Pila<String> s = new Pila<String>();
94
95         String elemento1 = "un texto";
96         String elemento2 = "otro elemento";
97         String elemento3 = "xxxx";
98
99         s.push(elemento1);
100        s.push(elemento2);
101        s.push(elemento3);
102
103        while (!s.estaVacia()) {
104            System.out.println(s.pop());
105            System.out.println("(" + s.getTamaño() + " elemento(s) quedan en la pila)");
106        }
107    }
108 }
```

Listing 207. Ejemplo de una pila genérica como lista ligada en Java.

23.5 Clases Genéricas en Scala⁵

Ejemplo:

```
1 class Stack[T] {  
2   var elems: List[T] = Nil  
3   def push(x: T) { elems = x :: elems }  
4   def top: T = elems.head  
5   def pop() { elems = elems.tail }  
6 }  
7  
8 object StackTest {  
9   def main(args: Array[String]) {  
10    val stack = new Stack[Int]  
11    stack.push(1)  
12    stack.push('a')  
13    println(stack.top)  
14    stack.pop()  
15    println(stack.top)  
16  }  
17 }
```

Listing 208. Ejemplo de clases genéricas en Scala.

⁵<https://docs.scala-lang.org/tour/generic-classes.html>

23.6 Clases Genéricas en C#⁷

Ejemplo:

⁷<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/>

23.7 Plantillas en D⁸

Ejemplo: Ejemplo: ⁹

```
1  class LinkedStack(T) {
2      private Node head;
3      class Node{
4          T item;
5          Node next;
6          Node prev;
7      }
8      this() {
9          head = new Node;
10         head.prev = head;
11         head.next = head;
12     }
13
14     void push(T t) {
15         Node temp = new Node;
16         temp.item = t;
17         head.prev.next = temp;
18         temp.prev = head.prev;
19         temp.next = head;
20         head.prev = temp;
21     }
22     T pop() {
23         Node temp = head.prev;
24         temp.prev.next = head;
25         head.prev = temp.prev;
26         return temp.item;
27     }
28     T peek() {
29         return head.prev.item;
30     }
31 }
32
33 void main() {
34     import std.stdio;
35 }
```

⁸<https://dlang.org/spec/template.html>

⁹https://en.wikibooks.org/wiki/A_Beginner%27s_Guide_to_D/Templates_and_Generic_Programming/Template_Classes

```
36  auto list = new LinkedStack!(int);
37  list.push(1);
38  list.push(2);
39  list.push(3);
40
41  writeln (list.pop());
42
43  auto strList = new LinkedStack!(string);
44  strList.push("hello");
45  strList.push("world!");
46
47  writeln(strList.pop());
48 }
```

Listing 209. Ejemplo de plantillas en D.

Ejercicios de clases genéricas

Diseñe una **clase genérica "Matriz"** que almacene una matriz de cualquier tipo. La clase debe tener métodos para acceder a un elemento específico en la matriz, para establecer el valor de un elemento específico en la matriz, para obtener el número de filas y columnas de la matriz, y para imprimir la matriz en la consola. Proporcione un ejemplo de cómo se utilizaría la clase para crear una matriz de enteros y una matriz de cadenas. Para C++ es posible hacer uso de la clase *vector* $< T >$ y para Java la clase *ArrayList* $< E >$

Implementar una **clase genérica *Arbol***, que permita almacenar elementos de cualquier tipo en una estructura de datos de tipo árbol binario. La clase debería tener los siguientes métodos:

1. *add(element : T)*: Agrega un elemento al árbol.
2. *remove(element : T)*: Elimina un elemento específico del árbol.
3. *search(element : T) – > bool*: Busca un elemento específico en el árbol y devuelve verdadero si se encuentra, falso en caso contrario.
4. *traverse(order : str) – > List[T]*: Recorre el árbol en el orden especificado (*in – order, pre – order, post – order*) y devuelve una lista con los elementos del árbol en ese orden.
5. *get_height() – > int*: devuelve la altura del arbol.
6. *get_size() – > int*: devuelve el número de elementos en el árbol.

Almacenamiento Genérico de Datos

Crea una **clase genérica llamada *AlmacenamientoDatosT*** que pueda almacenar y manipular datos de cualquier tipo *T*. La clase debe tener las siguientes funcionalidades:

1. **Inicialización**: La clase debe inicializarse con una capacidad inicial para almacenar elementos.
2. **Agregar Elemento**: Implementa un método *void agregarElemento(T elemento)* que añade un elemento al almacenamiento. Si el almacenamiento alcanza su capacidad, debería redimensionarse automáticamente para acomodar más elementos.
3. **Recuperar Elemento**: Implementa un método *T obtenerElemento(int indice)* que recupera el elemento en el índice especificado.
4. **Eliminar Elemento**: Implementa un método *bool eliminarElemento(T elemento)* que elimina la primera ocurrencia del elemento especificado del almacenamiento. Debería devolver *true* si se encuentra y elimina el elemento; de lo contrario, devolver *false*.

5. **Imprimir Todos los Elementos**: Implementa un método *void ImprimirTodos*

Capítulo 24

Manejo de Excepciones

Siempre se ha considerado importante el manejo de los errores en un programa, pero no fue hasta que surgió el concepto de **manejo de excepciones** que se dio una estructura más formal para hacerlo.

Concepto
El término de excepción viene de la posibilidad de detectar eventos que no forman parte del curso normal del programa, pero que de todas formas ocurren.

Un evento "**excepcional**" puede ser generado por una falla en la conexión a red, un archivo que no puede encontrarse, o un acceso indebido en memoria. La intención de una excepción es responder de manera dinámica a los errores, sin que afecte gravemente la ejecución de un programa, o que al menos se controle la situación posterior al error.

¿Cuál es la ventaja con respecto al manejo común de errores?

Normalmente, cada programador agrega su propio código de manejo de errores y queda revuelto con el código del programa. El manejo de excepciones indica claramente en que parte se encuentra el manejo de los errores, separándolo del código normal.

Además, es posible recibir y tratar muchos de los errores de ejecución y tratarlos correctamente, como podría ser una división entre cero.

Se recomienda el manejo de errores para aquellas situaciones en las cuales el programa necesita ayuda para recuperarse.

24.1 Manejo de Excepciones en C++

El manejo de excepciones en C++, involucra los siguientes elementos sintácticos:

try. El bloque definido por la instrucción *try*, especifica el código que potencialmente podría generar un error que deba ser manejado por la excepción:

Sintaxis
<pre>try { // instrucciones donde las excepciones // pueden ser generadas }</pre>

throw. Esta instrucción seguida por una expresión de un cierto tipo, genera una excepción del tipo de la expresión. Esta instrucción debería ser ejecutada dentro de algún bloque *try*, de manera directa o indirecta:

Sintaxis
<pre>throw "Se genera una excepción de tipo char *";</pre>

catch. La instrucción *catch* va seguida de un bloque *try*. *Catch* define un segmento de código para tratar una excepción (de un tipo) lanzada:

Sintaxis
<pre>catch (char *mensaje) { // instrucciones donde la excepción // thrown char * // será procesada }</pre>

Ejemplo:

```
1 // exceptions
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     try
7     {
8         throw 20;
9     }
10    catch (int e)
11    {
12        cout << "Una excepción ocurrió. Número: " << e << endl;
13    }
14    return 0;
15 }
```

Listing 210. Ejemplo de manejo de excepciones en C++.

24.1.1 Excepciones estandar en C++

La biblioteca estándar de C++ proporciona una clase base diseñada específicamente para declarar objetos que pueden ser lanzados como excepciones. La clase *exception* esta declarada en `< exception >` (en el espacio de nombres *std*). La clase tiene entre otras cosas un método virtual llamado *what* que regresa un arreglo de caracteres y puede ser redefinida en clases derivadas para describir la excepción.

Ejemplo:

```
1 // excepciones estándar
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 class myexception: public exception
7 {
8     virtual const char* what() const throw()
9     {
10         return "Mi excepción se ejecutó";
11     }
12 } myex;
13
14 int main () {
```

```
15  try
16  {
17      throw myex;
18  }
19  catch (exception& e)
20  {
21      cout << e.what() << endl;
22  }
23  return 0;
24 }
```

Listing 211. Ejemplo excepciones estandar en C++.

Las clases de la biblioteca estándar implementan clases derivadas de la clase *exception* para poder lanzar excepciones derivadas de esta clase.

Ejemplo:

```
1  // excepción bad_alloc
2  #include <iostream>
3  #include <exception>
4  using namespace std;
5
6  int main () {
7      try
8      {
9          int* myarray= new int[1000];
10     }
11     catch (exception& e)
12     {
13         cout << "Excepción estándar: " << e.what() << endl;
14     }
15     return 0;
16 }
```

Listing 212. Ejemplo excepción C++.

24.2 Manejo de Excepciones en Java

El modelo de excepciones de Java es similar al de C y C++, pero mientras en estos lenguajes no estamos obligados a manejar las excepciones, en Java es forzoso para el uso de ciertas clases; de lo contrario, el compilador generará un error.

24.2.1 ¿Cómo funciona?

Muchos tipos de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado hasta un intento de dividir por cero o intentar acceder a un arreglo fuera de sus límites. Cuando esto ocurre, la máquina virtual de Java crea un objeto de la clase *Exception* o *Error* y se notifica el hecho al sistema de ejecución. En este punto, se dice que se ha lanzado una excepción.

Un método se dice que es capaz de tratar una excepción si ha **previsto** el error que se ha producido y prevé también las operaciones a realizar para “recuperar” el programa de ese estado de error.

En el momento en que es lanzada una excepción, la máquina virtual de Java recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual de Java muestra un mensaje de error y el programa termina.

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción `throw`, lo que facilita la devolución de un código de error al método que invocó el método que causó el error.

Un **ejemplo** de una excepción generada (y no tratada) es el siguiente programa:

```
1 public class Excepcion {  
2     public static void main(String argumentos[]) {  
3         int i=5, j=0;  
4         int k=i/j; // División por cero  
5     }  
6 }
```

Listing 213. Ejemplo de excepción en Java.

Al ejecutarlo, se verá que la máquina virtual Java ha detecta una condición de error y ha crea un objeto de la clase *java.lang.ArithmeticException*. Como el método donde se ha producido la excepción no es capaz de tratarla, es manejada

por la máquina virtual Java, que muestra un mensaje de error y finaliza la ejecución del programa.

24.2.2 Lanzamiento de excepciones (*throw*)

Como se ha comentado anteriormente, un método también es capaz de lanzar excepciones.

Sintaxis

```
método ( ) throws <lista de excepciones> {  
    //código  
    ...  
    throw new <nombre Excepción>  
    ...  
}
```

donde *< listadeexcepciones >* es el nombre de cada una de las excepciones que el método puede lanzar.

Por ejemplo, en el siguiente programa se genera una condición de error si el dividendo es menor que el divisor:

Ejemplo:

```
1 public class LanzaExcepcion {  
2     public static void main(String argumentos[]) throws ArithmeticException {  
3  
4         int i=1, j=0;  
5         if (j==0)  
6             throw new ArithmeticException();  
7         else  
8             System.out.println(i/j);  
9     }  
10 }
```

Listing 214. Ejemplo lanzamiento de excepción en Java.

Para lanzar la excepción es necesario crear un objeto de tipo *Exception* o alguna de sus subclases (por ejemplo: *ArithmeticException*) y lanzarlo mediante la instrucción *throw*.

Los dos ejemplos vistos anteriormente, son capaces de lanzar una excepción en un momento dado, pero hasta aquí no difieren en mucho en su ejecución, ya que el resultado finalmente es la terminación del programa. En la siguiente sección se menciona como podemos darles un manejo especial a las excepciones, de tal forma que el resultado puede ser previsto por el programador.

24.2.3 Manejo de excepciones

En Java, de forma similar a C++ se pueden tratar las excepciones previstas por el programador utilizando unos mecanismos, los manejadores de excepciones, que se estructuran en tres bloques:

- El bloque *try*.
- El bloque *catch*.
- El bloque *finally*.

Concepto
Un manejador de excepciones es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar.

El bloque *try*

Lo primero que hay que hacer para que un método sea capaz de tratar una excepción generada por la máquina virtual Java o por el propio programa mediante una instrucción *throw*, es encerrar las instrucciones susceptibles de generarla en un bloque *try*.

```
1 try {  
2 <instrucciones>  
3 }  
4 ...
```

Cualquier excepción que se produzca dentro del bloque *try* será analizada por el bloque o bloques *catch* que se verá en el punto siguiente. En el momento en que se produzca la excepción, se abandona el bloque *try* y, por lo tanto, las instrucciones que sigan al punto donde se produjo la excepción no serán ejecutadas.

El bloque `catch`

Cada bloque *try* debe tener asociado por lo menos un bloque *catch*.

Sintaxis

```
try {  
    <instrucciones>  
} catch (TipoExcepción1 nombreVariable1) {  
    <instruccionesBloqueCatch1>  
} catch (TipoExcepción2 nombreVariable2) {  
    <instruccionesBloqueCatch2>  
}  
...  
catch (TipoExcepciónN nombreVariableN) {  
    <instruccionesBloqueCatchN>  
}
```

Por cada bloque *try* pueden declararse uno o varios bloques *catch*, cada uno de ellos capaz de tratar un tipo de excepción.

Para declarar el tipo de excepción que es capaz de tratar un bloque *catch*, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.

Ejemplo:

```
1 public class ExcepcionTratada {  
2     public static void main(String argumentos[]) {  
3         int i=5, j=0;  
4         try {  
5             int k=i/j;  
6             System.out.println("Esto no se va a ejecutar.");  
7         }  
8         catch (ArithmeticException ex) {  
9             System.out.println("Ha intentado dividir por cero");  
10        }  
11        System.out.println("Fin del programa");  
12    }  
13 }
```

Listing 215. Ejemplo de excepción tratada en Java.

La ejecución se resuelve de la siguiente forma:

1. Cuando se intenta dividir por cero, la máquina virtual Java genera un objeto de la clase *ArithmeticException*.
2. Al producirse la excepción dentro de un bloque *try*, la ejecución del programa se pasa al primer bloque *catch*.
3. Si la clase de la excepción se corresponde con la clase o alguna subclase de la clase declarada en el bloque *catch*, se ejecuta el bloque de instrucciones *catch* y a continuación se pasa el control del programa a la primera instrucción a partir de los bloques *try-catch*.

También se podría haber utilizado en la declaración del bloque *catch*, una superclase de la clase *ArithmeticException*.

Por ejemplo:

```
catch (RuntimeException ex)
```

o

```
catch (Exception ex)
```

Sin embargo, es mejor utilizar excepciones más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error y si tratan de capturar todas las excepciones de una forma muy general, posiblemente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

El bloque *finally*

El bloque *finally* se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, **incluso** si no se produce ninguna excepción.

Este bloque **garantiza** que el código que contiene será ejecutado independientemente de que se genere o no una excepción:

Sintaxis

```
try {  
    <instrucciones>  
}  
catch (TipoExcepción1 nombreVariable1) {  
    <instruccionesBloqueCatch1>  
}  
  
catch (TipoExcepción2 nombreVariable2) {  
    <instruccionesBloqueCatch2>  
}  
  
...  
catch (TipoExcepciónN nombreVariableN) {  
    <instruccionesBloqueCatchN>  
}  
finally {  
    <instruccionesBloqueFinally>  
}
```

Es utilizado para no tener que repetir código en el bloque *try* y en los bloques *catch*. Este código sirve para llevar a buen término el bloque de código independientemente del resultado.

Veamos ahora la clase *ExcepcionTratada* con el bloque *finally*. [Ejemplo:](#)

```
1 public class ExcepcionTratada {  
2     public static void main(String argumentos[]) {  
3         int i=5, j=0;  
4         try {  
5             int k=i /* /j */; //probar con y sin error  
6         }  
7         catch (ArithmeticException ex) {  
8             System.out.println("Ha intentado dividir por cero");  
9         }  
10        finally {  
11            System.out.println("Salida de finally");  
12        }  
13        System.out.println("Fin del programa");  
}
```

```

14     }
15 }

```

Listing 216. Ejemplo de excepción con *finally* en Java.

Un [ejemplo](#) derivando la clase `Exception` de Java en un estilo similar al uso de la clase correspondiente en C++:

```

1  class DivisionByZeroException extends Exception {
2      DivisionByZeroException(String msg) { super(msg); }
3  }
4
5  public class DivisionByZero {
6      public void division() throws DivisionByZeroException {
7          int num1 = 10;
8          int num2 = 0;
9
10         if (num2 == 0)
11             throw new DivisionByZeroException("/ entre 0");
12         System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
13         System.out.println("terminando division().");
14     }
15
16     public static void main(String args[]) {
17         try {
18             new DivisionByZero().division();
19         } catch (DivisionByZeroException e) {
20             System.out.println("En main, tratando con " + e);
21         } finally {
22             System.out.println("Finally ejecutado en main.");
23         }
24         System.out.println("Finalizando main.");
25     }
26 }

```

Listing 217. Ejemplo derivando de *Exception* en Java.

24.2.4 Jerarquía de excepciones

Las excepciones son objetos pertenecientes a la clase *Throwable* o alguna de sus subclases.

Dependiendo del lugar donde se produzcan existen dos tipos de excepciones:

1. Las excepciones **síncronas** no son lanzadas en un punto arbitrario del programa sino que, en cierta forma, son previsibles en determinados puntos del programa como resultado de evaluar ciertas expresiones o la invocación de determinadas instrucciones o métodos.
2. Las excepciones **asíncronas** pueden producirse en cualquier parte del programa y no son tan previsibles. Pueden producirse excepciones asíncronas debido a dos razones:
 - La invocación del método *stop()* de la clase *Thread* que se está ejecutando.
 - Un error interno en la máquina virtual Java.

Dependiendo de si el compilador comprueba o no que se declare un manejador para tratar las excepciones, se pueden dividir en:

1. Las excepciones **comprobables** son repasadas por el compilador Java durante el proceso de compilación, de forma que si no existe un manejador que las trate, generará un mensaje de error.
2. Las excepciones **no comprobables** son la clase *RuntimeException* y sus subclases junto con la clase *Error* y sus subclases.

También pueden definirse por el programador subclases de las excepciones anteriores. Las más interesantes desde el punto de vista del programador son las subclases de la superclase *Exception* ya que éstas pueden ser **comprobadas** por el compilador.

La jerarquía completa de excepciones existentes en el paquete *java.lang* se puede consultar más adelante¹.

24.2.5 Excepciones definidas por el usuario

A partir de la jerarquía de excepciones definidas, es posible para el programador especificar su propia excepción tomando como base alguna de las excepciones de la jerarquía. Por ejemplo:

```
1 class MiExcepcion extends Exception{
2     String cad;
3
4     /* El constructor de nuestra excepción copia a una cadena
5     el mensaje que se pasa al lanzar la excepción
6     */
```

¹Para un listado actual ver la documentación del jdk de Java más reciente.

```
7
8     MiExcepcion(String msj) {
9         cad=msj;
10    }
11    public String toString(){
12        return ("Se lanzó MiExcepcion: "+cad) ;
13    }
14 }
15
16 class EjemploMiExcepcion{
17     public static void main(String args[]){
18         try{
19             System.out.println("Iniciando bloque try");
20             // Lanzando mi propia excepción
21             throw new MiExcepcion("Mi mensaje de error");
22         }
23         catch (MiExcepcion exp) {
24             System.out.println("Bloque catch") ;
25             System.out.println(exp) ;
26         }
27     }
28 }
```

Listing 218. Ejemplo de excepción definida por el programador.

24.2.6 Ventajas del tratamiento de excepciones

Las ventajas, mencionadas por Díaz-Alejo², de un mecanismo de tratamiento de excepciones como este son varias:

- Separación del código "útil" del tratamiento de errores.
- Propagación de errores a través de la pila de métodos.
- Agrupación y diferenciación de errores mediante jerarquías.
- Claridad del código y obligación del tratamiento de errores.

²Díaz-Alejo Gómez, J.A., Programación con Java, IES Camp, Valencia, España, Last access: September 2006

24.2.7 Lista de Excepciones³

La jerarquía de clases derivadas de *Error* existentes en el paquete *java.lang* es la siguiente:

- `java.lang.Object`
 - `java.lang.Throwable` (implements `java.io.Serializable`)
- `java.lang.Error`
 - `java.lang.AssertionError`
 - `java.lang.LinkageError`
 - `java.lang.ClassCircularityError`
 - `java.lang.ClassFormatError`
 - `java.lang.UnsupportedClassVersionError`
 - `java.lang.ExceptionInInitializerError`
 - `java.lang.IncompatibleClassChangeError`
 - `java.lang.AbstractMethodError`
 - `java.lang.IllegalAccessError`
 - `java.lang.InstantiationError`
 - `java.lang.NoSuchFieldError`
 - `java.lang.NoSuchMethodError`
 - `java.lang.NoClassDefFoundError`
 - `java.lang.UnsatisfiedLinkError`
 - `java.lang.VerifyError`
 - `java.lang.ThreadDeath`
 - `java.lang.VirtualMachineError`
 - `java.lang.InternalError`
 - `java.lang.OutOfMemoryError`
 - `java.lang.StackOverflowError`
 - `java.lang.UnknownError`

La jerarquía de clases derivadas de *Exception* existentes en el paquete *java.lang* es la siguiente:

- `java.lang.Object`
 - `java.lang.Throwable` (implements `java.io.Serializable`)
 - `java.lang.Exception`
- `java.lang.ClassNotFoundException`
- `java.lang.CloneNotSupportedException`
- `java.lang.IllegalAccessException`
- `java.lang.InstantiationException`
- `java.lang.InterruptedException`
- `java.lang.NoSuchFieldException`

³Lista obtenida de la documentación del jdk en su versión 1.6

- `java.lang.NoSuchMethodException`
- `java.lang.RuntimeException`
 - `java.lang.ArithmeticException`
 - `java.lang.ArrayStoreException`
 - `java.lang.ClassCastException`
 - `java.lang.EnumConstantNotPresentException`
 - `java.lang.IllegalArgumentException`
 - `java.lang.IllegalThreadStateException`
 - `java.lang.NumberFormatException`
 - `java.lang.IllegalMonitorStateException`
 - `java.lang.IllegalStateException`
 - `java.lang.IndexOutOfBoundsException`
 - `java.lang.ArrayIndexOutOfBoundsException`
 - `java.lang.StringIndexOutOfBoundsException`
 - `java.lang.NegativeArraySizeException`
 - `java.lang.NullPointerException`
 - `java.lang.SecurityException`
 - `java.lang.TypeNotPresentException`
 - `java.lang.UnsupportedOperationException`

Las principales excepciones en otros paquetes Java son:

- `class java.lang.Object`
 - `class java.lang.Throwable`
 - `class java.lang.Error`
 - `java.awt.AWTError`
 - `class java.lang.Exception`
- `java.io.IOException`
- `java.io.EOFException`
- `java.io.FileNotFoundException`
- `java.io.InterruptedIOException`
- `java.io.UTFDataFormatException`
- `java.net.MalformedURLException`
- `java.net.ProtocolException`
- `java.net.SocketException`
- `java.net.UnknownHostException`
- `java.net.UnknownServiceException`
- `RuntimeException`
- `java.util.EmptyStackException`
- `java.util.NoSuchElementException`
- `java.awt.AWTException`

24.3 Manejo de Excepciones en Ruby

Ruby soporta el manejo de excepciones como cualquier lenguaje actual de programación. El manejo de excepciones en Ruby, involucra los siguientes elementos sintácticos:

24.3.1 *Raise & Rescue*

La ejecución de *raise* produce una excepción. *raise* es un método del módulo Kernel y tiene un alias definido como *fail*. La sintaxis se presenta a continuación:

Sintaxis
<pre>raise ExceptionClass[, "message"]</pre>

Ejemplo:

```
1 raise           # se relanza la ultima excepcion
2 raise "Danger, Will Robinson!"
3 raise "Houston, we have a problem"
4 raise ArgumentError, "Falla de datos"
5 raise ArgumentError.new("Falla de datos")
6 raise "Falta nombre" if nombre.nil?
```

Ahora, ¿Cómo agrupamos un segmento de código en el cual se quieren manejar excepciones? Podemos agrupar el código usando un bloque *begin end*. Dentro podemos incluir la cláusula *rescue*, por ejemplo:

Sintaxis
<pre>begin expr.. [rescue [tipo_de_error [=> var],...] expr..].. [else expr..] [ensure expr..] end</pre>

La sintaxis anterior implica que podemos poner una serie de cláusulas *rescue* especificando diferentes tipos de errores que pueden ser “rescatados” y la cláusula *else* recibiría aquellos errores que no entren dentro de los especificados por *rescue*.

La cláusula *ensure* es usada para especificar código que queremos que se ejecute independientemente del error generado. Por ejemplo:

```
1 begin
2   # Error...
3 rescue
4   # intento de recuperación...
5   retry # tratar de nuevo
6 ensure
7   # Este código es siempre ejecutado
8 end
```

```
1 def raise_exception
2   puts 'Antes de raise.'
3   raise 'Ocurrio un error'
4   puts 'Después de raise'
5 end
6 raise_exception
7
8 def raise_y_rescue
9   begin
10     puts 'Antes de raise.'
11     raise 'Ocurrio un error.'
12     puts 'Después de raise.'
13     rescue
14       puts 'Siendo rescatado.'
15     end
16     puts 'Despues del bloque begin - end.'
17   end
18   raise_y_rescue
19
20   begin
21     # ...
22     rescue UnaExepcion
23       # ...
24     rescue OtroTipoDeExepcion
25       # ...
```

```
26         else
27             # Otras excepciones
28         end
29
30     begin
31         raise "Probando excepciones."
32     rescue Exception => e
33         puts "Salida:"
34             puts e.message
35             puts e.backtrace.inspect
36             puts "fin salida."
37     end
38
39 x = a/b rescue puts("Division entre cero!")
```

Listing 219. Ejemplo de excepciones en Ruby.

24.3.2 Jerarquía de excepciones

Las excepciones en Ruby dependen de una jerarquía de herencia la cual tiene como superclase a *Exception*⁴:

```
* Exception
  o NoMemoryError
  o ScriptError
    + LoadError
    + NotImplementedError
    + SyntaxError
  o SignalException
    + Interrupt
  o StandardError (default for rescue)
    + ArgumentError
    + IOError
      # EOFError
    + IndexError
    + LocalJumpError
    + NameError
      # NoMethodError
    + RangeError
      # FloatDomainError
```

⁴Davis, R., Ruby QuickRef, 2004, <http://www.zenspider.com/Languages/Ruby/QuickRef.html>, Lastaccess:June2008

```

+ RegexpError
+ RuntimeError (default for raise)
+ SecurityError
+ SystemCallError
  # Errno::*
+ SystemStackError
+ ThreadError
+ TypeError
+ ZeroDivisionError
o SystemExit
o fatal

```

De igual forma es posible derivar algunas de las clases de la jerarquía de excepciones para crear una excepción más especializada.

24.3.3 *Catch & Throw*

Es posible en Ruby usar también los clásicos *catch & throw* los cuales son usados comúnmente cuando es necesario saltar de un punto de anidamiento más profundo[?].

Sintaxis

```

catch (:label) do
  #...
end

throw :label #salta atrás hasta el catch correspondiente y ejecuta el bloque

```

```

1 def pregunta pr
2   print pr
3   res = readline.chomp
4   throw :salida_solicitada if res == "!"
5   res
6 end
7
8 catch :salida_solicitada do
9   nombre = pregunta "Nombre: "

```

```
10 edad = pregunta "Edad: "  
11 sexo = pregunta "Sexo: "  
12 # ...  
13 end
```

Listing 220. Ejemplo *catch & throw* en Ruby.

24.4 Manejo de Excepciones en Python

Como la mayoría de los lenguajes actuales, Python soporta el manejo de excepciones. Un resumen de la forma en que funcionan se muestra a continuación:

- *try/except*. Capturar y recuperarse de excepciones levantadas por Python o por nuestro código.
- *try/finally*. Ejecutar acciones de limpieza ya sea que ocurran o no excepciones.
- *raise*. Lanzar una excepción manualmente en nuestro código.

De manera similar a otros lenguajes, podemos ver excepciones que son generadas por diversas situaciones y al no ser manejadas implican la interrupción de la ejecución del programa.

```
1  #Ejemplo de excepción lanzada por un error en código
2
3  i=5
4  j=0
5  k=i/j
```

Listing 221. Ejemplo de excepción lanzada en python.

En el siguiente ejemplo vemos el uso de *raise* para lanzar una excepción manualmente en nuestro código.

```
1  #ejemplo de lanzamiento de excepción sin tratamiento
2
3  i=1
4  j=0
5  if j==0 :
6      raise ZeroDivisionError('División entre 0')
7  else:
8      print(i/j)
```

Listing 222. Ejemplo de lanzamiento de excepción sin tratamiento en Python.

De manera similar a los otros lenguajes, lo adecuado es manejar las posibles excepciones mediante bloques especiales. La sintaxis general es:

Sintaxis

```
try:
    <instrucciones>
except [<tipo> [as <variable>]]:
    <instrucciones>
[except [<tipo> [as <variable>]]:
    <instrucciones>]*
[else:
    <instrucciones>]
[finally:
    <instrucciones>]
```

En el siguiente ejemplo, vemos como se maneja la división entre cero con el bloque *try/except*.

```
1  #tratamiento de excepciones
2
3  i=5
4  j=0
5
6  try:
7      k=i/j
8      print('Esto no se va a ejecutar')
9  except ZeroDivisionError:
10     print('Ha intentado dividir por cero')
11
12 print('fin del programa')
```

Listing 223. Ejemplo de tratamiento de excepciones en Python.

En el siguiente agregamos el bloque *finally*, que se ejecuta independientemente de que se genere o no la excepción.

```
1  #tratamiento de excepciones y tratamiento general con finally
2
3  i=5
4  j=0
5
6  try:
```



```
7     k=i/j
8     print('Esto no se va a ejecutar')
9     except ZeroDivisionError:
10         print('Ha intentado dividir por cero')
11     finally:
12         print('Salida de finally')
13
14 print('fin del programa')
```

Listing 224. Ejemplo de tratamiento de excepciones y tratamiento general con finally en Python.

También podemos definir nuestra propia jerarquía de excepciones si así lo consideramos necesario.

```
1  #Excepción definida por el usuario
2
3  class DivisionByZeroException (Exception):
4      def __str__(self):
5          return 'Mi excepcion'
6
7  class DivisionByZero:
8      def division(self):
9          num1=10
10         num2=0
11
12         if num2==0:
13             raise DivisionByZeroException()
14
15         print('División: ' + str(num1/num2))
16
17  #script
18  try:
19      DivisionByZero().division()
20  except DivisionByZeroException:
21      print('Ejecutando script, detectando excepción')
22  finally:
23      print('Finally siendo ejecutado en el script de prueba')
24
25  print('Finalizando main')
```

Listing 225. Ejemplo de excepción definida por el usuario en Python.

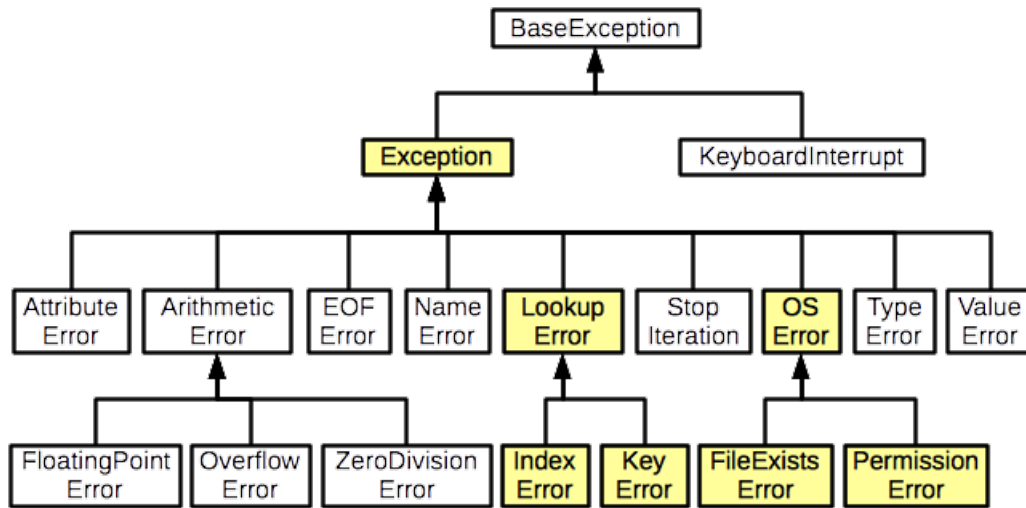


Fig. 24.1. Jerarquía de herencia de las excepciones en Python

Cuadro 24.1. Resumen de excepciones de Python

Opción	Interpretación
except:	Recibe todas las excepciones. O todas las demás si ya se recibieron otras antes.
except <tipo>:	Recibe una excepción en particular.
except <tipo>as <variable>:	Recibe la excepción y asigna su instancia
except (<tipo1>, <tipo2>):	Recibe todas las excepciones indicadas.
except (<tipo1>, <tipo2>) as <variable>	Recibe todas las excepciones indicadas y asigna su instancia a una variable.
else:	Se ejecuta si no es lanzada ninguna excepción en el bloque try.
finally:	Siempre se ejecuta al salir del segmento de manejo de excepciones.

Toda excepción que creamos debe ser una subclase de *Exception*. La figura ?? muestra la jerarquía de herencia de las excepciones de Python.

Un resumen se puede ver en el cuadro ??.

Un ejemplo resumiendo las opciones en un solo programa:

```

1  # Ejemplo try/except/else/finally
2  sep = '-' * 45 + '\n'
3  print(sep + 'Excepción lanzada y catchada')
4  try:
5      x = 'spam'[99]
6  except IndexError as e:
7      print('except: ', e)
8  finally:
9      print('finally')
10 print('después de bloque try')

```

```
11
12 print(sep + 'NO se lanza excepción')
13 try:
14     x = 'spam'[3]
15 except IndexError:
16     print('except')
17 finally:
18     print('finally')
19 print('después de bloque try')
20
21 print(sep + 'NO se lanza excepción, con opción else')
22 try:
23     x = 'spam'[3]
24 except IndexError:
25     print('except')
26 else:
27     print('else')
28 finally:
29     print('finally')
30 print('after')
31
32 print(sep + 'Excepción lanzada pero no catchada')
33 try:
34     x=1 /0
35 except IndexError:
36     print('except')
37 finally:
38     print('finally')
39 print('after')
```

Listing 226. Ejemplo resumiendo opciones de manejo de excepciones en Python.

Parte IV

Más allá de los Objetos

Capítulo 25

Afirmaciones

Las afirmaciones son usadas para verificar **invariantes** en un programa [?]. Es una manera simple de probar una condición que **siempre** debe ser verdadera. Si la afirmación resulta ser falsa se considera un error y se interrumpe la ejecución. Escribir afirmaciones mientras se programa es una de las más rápidas y efectivas formas de detectar y corregir errores [?].

Las afirmaciones por lo tanto son usadas para comprobar código que se asume será verdadero, siendo la afirmación la parte responsable de verificar que realmente es verdadero.

Las afirmaciones pueden ser utilizadas como una aproximación de la técnica de **diseño por contrato**. Podemos usar afirmaciones para definir¹:

- Precondiciones. Predicados que deben ser verdaderos cuando un método es invocado.
- Postcondiciones. Predicados que deben ser verdaderos después de la ejecución exitosa de un método.
- Invariantes de clase. Predicados que deben ser verdaderos para cada instancia de una clase.

¹Java assert

25.1 Afirmaciones en C++

Las afirmaciones en C++ se manejan a través de la macro²:

Sintaxis
<code>assert (int expression)</code>

Que se encuentra en la biblioteca *assert.h*. Si la expresión dentro de *assert* es falsa, se interrumpirá la ejecución del programa.

Las afirmaciones serán deshabilitadas si previo a la introducción de la biblioteca se introduce la instrucción de preprocesador definiendo la macro *NDEBUG*:

```
#define NDEBUG
```

```
1  /* Ejemplo assert */
2  #include <stdio.h>
3  #include <assert.h>
4
5  void print_number(int* myInt) {
6      assert (myInt!=NULL);
7      printf ("%d\n", *myInt);
8  }
9
10 int main () {
11     int a=10;
12     int * b = NULL;
13     int * c = NULL;
14
15     b=&a;
16     print_number (b);
17     print_number (c);
18
19     return 0;
20 }
```

Listing 227. Ejemplo de afirmaciones en C++.

²C++ assert

25.2 Afirmaciones en Java

Las afirmaciones fueron introducidas en Java desde la versión 1.4 del jdk. Cada afirmación debe contener una expresión booleana (*boolean* o *Boolean*).

Sintaxis
<code>assert Expression1;</code> o <code>assert Expression1 : Expression2 ;</code>

donde *Expression1* es una expresión booleana. Esta expresión es la evaluada y si es falsa la excepción *AssertionError* es lanzada. *Expression2* es una expresión que devuelve un valor (no *void*) que generalmente es usado para proveer de un mensaje para la excepción *AssertionError*.

25.2.1 Usando afirmaciones

Es importante no introducir código en las afirmaciones que en realidad sea una acción del programa. Por ejemplo:

```
assert ++i < max;
```

Es inapropiado pues se esta modificando el estado del programa al mismo tiempo que validando. Lo correcto sería algo del estilo:

```
1 i++;  
2 assert i < max;
```

Errores detectados con afirmaciones deben ser errores que no deben pasar. Es por esto que se lanza un subtipo de *Error* en lugar de un subtipo de *Exception*. Si falla la validación de una afirmación se asume un error grave que nunca debe pasar.

25.2.2 Habilitando y deshabilitando las afirmaciones

Por omisión, las afirmaciones están deshabilitadas en tiempo de ejecución. Para cambiar de un estado a otro deben aplicarse parámetros especiales en la ejecución de la máquina virtual de Java:

```
-enableassertions | -ea  
-disableassertions | -da
```

Estos modificadores pueden no llevar a su vez argumentos, por lo que active o desactiva las afirmaciones para todas las clases, o pueden indicarse nombres de paquetes ó clases específicas:

```
-enableSystemassertions | -esa  
-disableSystemassertions | -dsa.
```

Ejemplo:

```
1  //Recuerda habilitar el uso de afirmaciones  
2  public class AssertionEjemplo {  
3      public static void main(String argv[]) {  
4          //obtener un número del primer argumento  
5          int num = Integer.parseInt(argv[0]);  
6  
7          assert num <=10; //se detiene si num>10  
8  
9          System.out.println("Pasó");  
10     }  
11 }
```

Listing 228. Ejemplo de afirmaciones en Java.

25.3 Afirmaciones en Python

Python también soporta la instrucción *assert* para lanzar una excepción de manera condicional en nuestro código. La sintaxis es la siguiente:

Sintaxis
assert <expresión> [, 'mensaje opcional']

Que en realidad equivale a lanzar la excepción

```
raise AssertionError()
```

bajo una condición.

Ejemplo:

```
1  #Ejemplo de assert
2
3  def printNumber(myInt):
4      assert myInt!=None, 'Dato no debe ser nulo'
5      print(myInt)
6
7  a=10
8  b=None
9  c=None
10
11 b=a
12 printNumber(b)
13 printNumber(c)
```

Listing 229. Ejemplo de afirmaciones en Python.

Para descartar la ejecución de las afirmaciones se tiene que ejecutar el Python con la opción en la línea de comandos de `-O` (activando modo de Optimización), poniendo la variable `__debug__` en falso³:

```
$ python -O
```

³ [Assert statement](#)

25.4 Pruebas de unidad en Python

Las pruebas unitarias, también conocidas como unit testing en inglés, son una práctica fundamental en el desarrollo de software que se utiliza para verificar que las unidades individuales de código, como funciones o métodos, funcionen correctamente de forma aislada. Estas pruebas se centran en evaluar pequeñas partes del código de manera independiente, lo que facilita la identificación temprana de errores y garantiza que cada componente del programa se comporte como se espera.

Las pruebas unitarias son esenciales para asegurar la calidad y la robustez del código Python. Al realizar pruebas unitarias, se pueden detectar y corregir problemas antes de que se propaguen a otras partes del sistema. Esto promueve una mayor confiabilidad y mantenibilidad del código, ya que los errores se encuentran y se corrigen de manera más eficiente.

En Python, existen varios módulos y bibliotecas que se utilizan comúnmente para realizar pruebas unitarias y pruebas de software. Algunos de los módulos de pruebas más populares son:

- **unittest**: Este es el módulo de pruebas unitarias estándar en la biblioteca estándar de Python. Está inspirado en el marco de pruebas JUnit de Java y proporciona un conjunto completo de herramientas para escribir y ejecutar pruebas unitarias.
- **pytest**: pytest es una biblioteca de pruebas muy popular que ofrece una sintaxis simple y poderosa para escribir pruebas. Se destaca por su capacidad para descubrir y ejecutar automáticamente pruebas en un proyecto, lo que hace que la configuración sea más sencilla.

25.4.1 Pytest

Pytest es un marco de pruebas unitarias en Python que simplifica la escritura y ejecución de pruebas. A diferencia de la biblioteca de pruebas estándar `unittest`, pytest ofrece una sintaxis más sencilla y expresiva para crear pruebas, lo que facilita su adopción y uso. Además, pytest proporciona una amplia gama de características avanzadas y complementos para abordar diferentes situaciones de prueba.

Ventajas de pytest:

1. **Sintaxis Clara**: La sintaxis de pytest es simple y legible, lo que permite escribir pruebas de manera más eficiente y comprensible.
2. **Descubrimiento Automático de Pruebas**: pytest puede descubrir y ejecutar automáticamente pruebas en todo un proyecto sin requerir una configuración extensa. Esto ahorra tiempo y esfuerzo en la configuración de pruebas.

3. Amplia Compatibilidad: Pytest es compatible con otros marcos de pruebas como unittest, lo que facilita la migración de pruebas existentes.
4. Parametrización: Permite ejecutar la misma prueba con múltiples conjuntos de datos, lo que simplifica las pruebas con diferentes casos de entrada.

Ejemplos de Prueba Unitaria con pytest en Python

Supongamos que estamos desarrollando una función simple que suma dos números en Python. Primero, escribimos la función en el archivo *calculadora.py*:

```
1 # calculadora.py
2
3 def suma(a, b):
4     return a + b
```

Ejemplo:

```
1
2 # test_calculadora.py
3
4 import calculadora
5
6 def test_suma():
7     resultado = calculadora.suma(2, 3)
8     assert resultado == 5, "La suma de 2 y 3 debería ser 5"
9
10
```

Listing 230. Ejemplo de prueba de unidad con módulo pytest en Python.
En este ejemplo:

- Importamos el módulo *calculadora* que contiene la función *suma*.
- Definimos una función llamada *test_suma* que verifica si la función *suma* devuelve el resultado esperado cuando se le dan los argumentos 2 y 3.
- Usamos la afirmación (*assert*) para verificar si el resultado es igual a 5 y proporcionamos un mensaje de error personalizado si la afirmación falla.

Para ejecutar estas pruebas con pytest, simplemente abre una terminal en la ubicación de tus archivos y ejecuta el siguiente comando:

```
pytest test_calculadora.py
```

Otro ejemplo:

Supongamos que tenemos una función simple que calcula el área de un rectángulo en un archivo llamado `geometria.py`:

```
1 # geometria.py
2
3 def area_rectangulo(base, altura):
4     if base <= 0 or altura <= 0:
5         raise ValueError("La base y la altura deben ser números positivos.")
6     return base * altura
7
```

Ahora, creemos pruebas unitarias para esta función utilizando `pytest` en un archivo llamado `test_geometria.py`:

Ejemplo:

```
1 # test_geometria.py
2
3 import pytest
4 import geometria
5
6 def test_area_rectangulo():
7     resultado = geometria.area_rectangulo(3, 4)
8     assert resultado == 12, "El área del rectángulo con base 3 y altura 4 debe ser 12"
9
10 def test_area_rectangulo_excepcion():
11     with pytest.raises(ValueError):
12         geometria.area_rectangulo(-1, 5)
13
```

Listing 231. Ejemplo de prueba de unidad con módulo `pytest` en Python.
En el código de prueba:

- Importamos `pytest` y el módulo `geometria`, que contiene la función que estamos probando.
- Creamos dos funciones de prueba, `test_area_rectangulo` y `test_area_rectangulo_excepcion`, que verifican si la función `area_rectangulo` se comporta como se espera en diferentes situaciones.

- Utilizamos *assert* para verificar si los resultados son los esperados.
- En *test_area_rectangulo_excepcion*, usamos *pytest.raises* para verificar que la función eleve una excepción cuando se le proporciona una entrada inválida.

Con esto, *pytest* descubrirá automáticamente las pruebas en el archivo y proporcionará información detallada sobre su ejecución.

pytest es una herramienta esencial en el desarrollo de software en Python que simplifica la escritura y ejecución de pruebas unitarias. Su sintaxis clara y sus características avanzadas lo convierten en una elección poderosa para garantizar la calidad del código.

25.4.2 unittest

unittest es un módulo incorporado en Python que ofrece un marco de trabajo para la creación y ejecución de pruebas unitarias. Este módulo se inspira en el marco de pruebas JUnit de Java y sigue el enfoque de la orientación a objetos para organizar las pruebas. Permite a los desarrolladores crear pruebas efectivas al proporcionar una estructura coherente y herramientas para verificar el comportamiento de las funciones y métodos.

Características Clave de *unittest*:

1. Organización de Pruebas: *unittest* fomenta la organización de pruebas en clases, lo que facilita la gestión de conjuntos de pruebas relacionadas.
2. Métodos de Aserción: Ofrece una variedad de métodos de aserción, como *assertEqual*, *assertTrue*, *assertFalse*, entre otros, para verificar resultados esperados.
3. Descubrimiento Manual de Pruebas: A diferencia de algunos otros marcos de pruebas, *unittest* no ofrece descubrimiento automático de pruebas. Los desarrolladores deben especificar manualmente las pruebas a ejecutar.
4. Fixture: Permite la configuración y limpieza compartida para las pruebas a través de métodos especiales de configuración (*setUp*) y limpieza (*tearDown*).

Ejemplos de Prueba Unitaria con *unittest* en Python

Supongamos que estamos desarrollando una función simple que suma dos números en Python. Primero, escribimos la función en el archivo *calculadora.py*:

```
1 # calculadora.py
2
```

```
3 def suma(a, b):  
4     return a + b
```

A continuación, creamos un archivo de prueba unitaria llamado *test_calculadora.py* para verificar si la función *suma* funciona correctamente:

Ejemplo:

```
1 # test_calculadora.py  
2  
3 import unittest  
4 from calculadora import suma  
5  
6 class TestCalculadora(unittest.TestCase):  
7  
8     def test_suma(self):  
9         resultado = suma(2, 3)  
10        self.assertEqual(resultado, 5, "La suma de 2 y 3 debería ser 5")  
11  
12 if __name__ == '__main__':  
13     unittest.main()  
14
```

Listing 232. Ejemplo de prueba de unidad con módulo *unittest* en Python. En este ejemplo:

- Importamos el módulo *unittest* para crear nuestras pruebas unitarias.
- Importamos la función *suma* desde el módulo *calculadora*.
- Creamos una clase llamada *TestCalculadora* que hereda de *unittest.TestCase*. Esto nos permite definir métodos de prueba.
- Dentro de la clase *TestCalculadora*, definimos un método llamado *test_suma*. Este método verifica si la función *suma* devuelve el resultado esperado cuando se le dan los argumentos 2 y 3.
- Usamos el método *self.assertEqual* para comparar el resultado con el valor esperado.

Para ejecutar estas pruebas unitarias, simplemente ejecutamos el archivo *test_calculadora.py*. Si la función *suma* se comporta como se espera, no se mostrarán errores.

Para el ejemplo de *geometria.py* visto anteriormente el código de prueba en *unittest* queda como sigue:

Ejemplo:


```
1  # test_geometria.py
2
3  import unittest
4  import geometria
5
6  class TestGeometria(unittest.TestCase):
7
8      def test_area_rectangulo(self):
9          resultado = geometria.area_rectangulo(3, 4)
10         self.assertEqual(resultado, 12,
11             "El área del rectángulo con base 3 y altura 4 debe ser 12")
12
13         def test_area_rectangulo_excepcion(self):
14             with self.assertRaises(ValueError):
15                 geometria.area_rectangulo(-1, 5)
16
17 if __name__ == '__main__':
18     unittest.main()
19
```

Listing 233. Ejemplo de prueba de unidad con módulo unittest en Python.
En este código de prueba:

- Importamos el módulo unittest y el módulo geometria que contiene la función que estamos probando.
- Creamos una clase llamada TestGeometria que hereda de unittest.TestCase, lo que nos permite definir métodos de prueba.
- Dentro de la clase TestGeometria, definimos dos métodos de prueba, `test_area_rectangulo` y `test_area_rectangulo_excepcion`.
Para ejecutar estas pruebas con *unittest*, simplemente ejecutamos el archivo `test_geometria.py`.
unittest es un módulo importante para realizar pruebas unitarias en Python. Su enfoque orientado a objetos y sus métodos de aserción facilitan la creación de pruebas estructuradas y confiables para garantizar que las unidades individuales de código funcionen correctamente.

25.5 Manejo de archivos: de texto, JSON y CSV

25.5.1 Manejo de archivos de texto

el manejo de archivos de texto se refiere a la capacidad de leer y escribir información en archivos de texto plano. Los archivos de texto son una forma común de almacenar datos estructurados de manera legible por humanos, lo que los hace ideales para tareas como guardar configuraciones, registros de actividad o información tabular. A continuación, se proporcionan ejemplos de cómo manejar archivos de texto en Python, centrándonos en las operaciones básicas de lectura y escritura.

Lectura de Archivos de Texto

Para leer información de un archivo de texto en Python, primero debemos abrir el archivo en modo lectura ('r'). Luego, podemos usar ciclos o métodos de lectura para procesar su contenido. Aquí hay un ejemplo:

Ejemplo:

```
1
2 # Abrir un archivo en modo lectura
3 with open('archivo.txt', 'r') as archivo:
4     # Leer todo el contenido del archivo
5     contenido = archivo.read()
6
7 # Imprimir el contenido leído
8 print(contenido)
9
```

Listing 234. Ejemplo de lectura de archivo de texto.

En este ejemplo, utilizamos la declaración *with* para garantizar que el archivo se cierre correctamente después de su uso. El método *read()* se emplea para leer todo el contenido del archivo y almacenarlo en la variable *contenido*.

Escritura en Archivos de Texto

Para escribir en un archivo de texto, debemos abrirlo en modo escritura ('w'). Si el archivo no existe, se creará; si existe, su contenido se sobrescribirá. A continuación, se muestra un ejemplo de escritura en un archivo de texto:

Ejemplo:

```
1
2 # Abrir un archivo en modo escritura
3 with open('nuevo_archivo.txt', 'w') as archivo:
```

```
4 # Escribir datos en el archivo
5 archivo.write("Este es un ejemplo de escritura en un archivo de texto.\n")
6 archivo.write("Python es un lenguaje de programación poderoso.\n")
7
```

Listing 235. Ejemplo de escritura de archivo de texto.

En este caso, utilizamos *with* nuevamente para asegurarnos de que el archivo se cierre correctamente después de escribir en él. Los datos se escriben en el archivo utilizando el método *write()*. Nota que agregamos el carácter de nueva línea ("*\n*") para separar las líneas.

Lectura Línea por Línea

Si deseamos leer un archivo línea por línea, podemos usar un ciclo *for*. Esto es especialmente útil para archivos grandes, ya que no cargamos todo el contenido en la memoria al mismo tiempo:

Ejemplo:

```
1
2 # Abrir un archivo en modo lectura
3 with open('archivo_grande.txt', 'r') as archivo:
4     for linea in archivo:
5         # Procesar cada línea
6         print(linea.strip()) # strip()
7         elimina espacios en blanco y saltos de línea al final
8
```

Listing 236. Ejemplo de lectura línea por línea de archivo de texto.

En este ejemplo, el ciclo *for* itera sobre cada línea del archivo, y utilizamos *strip()* para eliminar espacios en blanco y saltos de línea al final de cada línea.

El manejo de archivos de texto en Python es fundamental para leer y escribir datos de manera eficiente y efectiva. Con las operaciones básicas de lectura y escritura, los programadores pueden manipular archivos de texto de diversas maneras, lo que resulta esencial en muchas aplicaciones y tareas de procesamiento de datos.

25.5.2 Manejo de archivos JSON

Los archivos JSON (JavaScript Object Notation) son una forma eficiente y ampliamente utilizada para el intercambio de datos estructurados entre aplicaciones. A continuación, se proporcionarán ejemplos detallados de cómo manejar archivos JSON en Python, ilustrando su relevancia en el ámbito de la programación.

Lectura de Archivos JSON

La lectura de archivos JSON en Python es un proceso esencial cuando se necesita acceder a datos estructurados almacenados en este formato. Para llevar a cabo esta tarea, se utiliza la biblioteca estándar `json`. A continuación, se presenta un ejemplo de lectura de un archivo JSON:

Ejemplo:

```
1
2 import json
3
4 # Abrir y leer un archivo JSON en modo lectura
5 with open('datos.json', 'r') as archivo_json:
6     # Cargar los datos desde el archivo
7     datos = json.load(archivo_json)
8
9 # Acceder a los datos y realizar operaciones
10 print("Nombre:", datos["nombre"])
11 print("Edad:", datos["edad"])
12
```

Listing 237. Ejemplo de lectura de archivo JSON.

En este ejemplo, se utiliza `json.load()` para cargar los datos desde el archivo JSON en la variable `datos`. Luego, se pueden acceder y procesar los datos como diccionarios de Python.

Escritura en Archivos JSON

La escritura en archivos JSON es igualmente fundamental, ya que permite almacenar datos estructurados en un formato que es fácilmente comprensible por otros programas. Para llevar a cabo esta tarea, también empleamos la biblioteca `json`. Aquí se presenta un ejemplo de escritura en un archivo JSON:

Ejemplo:

```
1
2 import json
3
4 # Datos que se desean escribir en el archivo JSON
5 nuevos_datos = {
6     "nombre": "Ana",
7     "edad": 28,
8     "ciudad": "Ejemploville"
```

```
9 }
10
11 # Abrir y escribir en un archivo JSON en modo escritura
12 with open('nuevos_datos.json', 'w') as archivo_json:
13     # Escribir los datos en el archivo
14     json.dump(nuevos_datos, archivo_json)
15
16 print("Datos escritos en el archivo JSON.")
17
```

Listing 238. Ejemplo de escritura de archivo JSON.

En este caso, se utiliza *json.dump()* para escribir los datos en el archivo JSON. Estos datos pueden ser un diccionario de Python, como se muestra en el ejemplo.

Ventajas del Uso de Archivos JSON

El manejo de archivos JSON en Python es crucial para la interoperabilidad de datos entre sistemas y aplicaciones. JSON es un formato ligero y fácil de leer, lo que lo hace ideal para la transferencia y el almacenamiento de datos estructurados. Además, Python proporciona herramientas integradas para trabajar con archivos JSON de manera eficiente, lo que simplifica las tareas de lectura y escritura de datos.

25.5.3 Manejo de archivos CSV

Los archivos CSV son una forma común de almacenar datos tabulares en un formato que es fácilmente legible y editable tanto por humanos como por máquinas. A continuación, se presentarán ejemplos completos de cómo manejar archivos CSV en Python, destacando su relevancia en el ámbito de la programación y el procesamiento de datos.

Lectura de Archivos CSV

La lectura de archivos CSV en Python es esencial para analizar y procesar datos tabulares provenientes de diversas fuentes, como hojas de cálculo o bases de datos. Para llevar a cabo esta tarea, se utiliza la biblioteca estándar *csv*. A continuación, se presenta un ejemplo de lectura de un archivo CSV:

Ejemplo:

```
1
2 import csv
3
```

```
4 # Abrir y leer un archivo CSV en modo lectura
5 with open('datos.csv', 'r', newline='') as archivo_csv:
6     lector_csv = csv.reader(archivo_csv)
7
8     # Iterar a través de las filas del archivo CSV
9     for fila in lector_csv:
10         # Procesar cada fila (que es una lista de valores)
11         print("Nombre:", fila[0])
12         print("Edad:", fila[1])
13
```

Listing 239. Ejemplo de lectura de archivo CSV.

En este ejemplo, se utiliza `csv.reader()` para crear un objeto que nos permite iterar a través de las filas del archivo CSV. Cada fila se convierte en una lista de valores que podemos procesar según sea necesario.

Escritura en Archivos CSV

La escritura en archivos CSV es igualmente importante, ya que nos permite almacenar datos tabulares en un formato que puede ser compartido y utilizado por otros programas. También utilizamos la biblioteca `csv` para esta tarea. Aquí se presenta un ejemplo de escritura en un archivo CSV:

Ejemplo:

```
1
2 import csv
3
4 # Datos que se desean escribir en el archivo CSV
5 nuevos_datos = [
6     ["Ana", 28],
7     ["Carlos", 35],
8     ["Elena", 22]
9 ]
10
11 # Abrir y escribir en un archivo CSV en modo escritura
12 with open('nuevos_datos.csv', 'w', newline='') as archivo_csv:
13     escritor_csv = csv.writer(archivo_csv)
14
15     # Escribir los datos en el archivo CSV
16     for fila in nuevos_datos:
17         escritor_csv.writerow(fila)
18
```

```
19 print("Datos escritos en el archivo CSV.")
```

```
20
```

Listing 240. Ejemplo de escritura de archivo CSV.

En este caso, utilizamos `csv.writer()` para crear un objeto que nos permite escribir datos en el archivo CSV. Usamos el método `writerow()` para escribir cada fila de datos.

Ventajas del Uso de Archivos CSV en Python

El manejo de archivos CSV en Python es crucial para la manipulación y análisis de datos tabulares, lo que es esencial en una amplia variedad de aplicaciones, desde procesamiento de datos hasta análisis estadístico. Los archivos CSV son ampliamente compatibles y su estructura simple los hace ideales para el intercambio de datos entre diferentes sistemas. Python proporciona herramientas eficientes y flexibles para trabajar con archivos CSV, lo que simplifica las tareas de lectura y escritura de datos tabulares.

25.6 Casos de estudio orientados al aprendizaje computacional

25.6.1 ¿Qué es el aprendizaje computacional?

El aprendizaje computacional es una rama de la inteligencia artificial que se ocupa de la construcción de sistemas capaces de aprender a partir de datos. Estos sistemas, también conocidos como modelos de aprendizaje automático, pueden ser utilizados para realizar una amplia gama de tareas, como la clasificación de imágenes, la predicción de resultados y la detección de patrones. El aprendizaje computacional se basa en la idea de que los sistemas pueden aprender a realizar tareas sin ser explícitamente programados para ello. En lugar de ello, los sistemas aprenden a partir de datos de entrenamiento, que son ejemplos de cómo se debe realizar la tarea. Por ejemplo, un sistema de aprendizaje automático para clasificar imágenes podría ser entrenado con un conjunto de datos de imágenes etiquetadas, cada una de las cuales tiene un nombre de objeto. El sistema aprendería a identificar los patrones que distinguen a cada objeto, y luego podría utilizar esos patrones para clasificar nuevas imágenes. El aprendizaje computacional tiene una amplia gama de aplicaciones, que incluyen:

- **Reconocimiento de imágenes y voz:** Los sistemas de aprendizaje automático se utilizan para reconocer rostros, objetos y palabras.

- **Predicción:** Los sistemas de aprendizaje automático se pueden utilizar para predecir el comportamiento futuro, como el riesgo de que un cliente abandone una empresa o la probabilidad de que un paciente desarrolle una enfermedad.
- **Robótica:** Los sistemas de aprendizaje automático se utilizan para que los robots puedan aprender a realizar tareas en entornos complejos.
- **Médico:** Los sistemas de aprendizaje automático se utilizan para diagnosticar enfermedades, personalizar tratamientos y desarrollar nuevos fármacos.

El aprendizaje computacional es una disciplina en rápido desarrollo, y se espera que tenga un impacto cada vez mayor en nuestras vidas. Tipos de aprendizaje computacional Existen tres tipos principales de aprendizaje computacional:

- **Aprendizaje supervisado:** En el aprendizaje supervisado, el sistema de aprendizaje automático se proporciona con datos de entrenamiento que están etiquetados con la respuesta correcta. El sistema aprende a identificar los patrones que distinguen a cada respuesta correcta, y luego puede utilizar esos patrones para predecir la respuesta correcta para nuevos datos.
- **Aprendizaje no supervisado:** En el aprendizaje no supervisado, el sistema de aprendizaje automático no se proporciona con datos de entrenamiento etiquetados. El sistema aprende a identificar los patrones en los datos, y luego puede utilizar esos patrones para agrupar los datos en grupos o para identificar clusters.
- **Aprendizaje por refuerzo:** En este enfoque, un agente de aprendizaje interactúa con un entorno y toma decisiones para maximizar una recompensa acumulada a lo largo del tiempo. A medida que el agente interactúa con el entorno, aprende a tomar decisiones óptimas para lograr sus objetivos.

25.6.2 Bibliotecas de Python usadas en aprendizaje computacional

Python es un lenguaje de programación de propósito general que es cada vez más popular en el campo del aprendizaje computacional. Esto se debe a que Python es un lenguaje fácil de aprender y usar, y tiene una amplia gama de bibliotecas y herramientas disponibles para el aprendizaje automático.

Las principales bibliotecas de Python usadas en aprendizaje computacional son:

- **Scikit-learn:** Es una biblioteca de aprendizaje automático de código abierto que proporciona una amplia gama de algoritmos para el aprendizaje supervisado, no supervisado y refuerzo.

- **TensorFlow:** Es una biblioteca de aprendizaje profundo de código abierto que es popular para el desarrollo de redes neuronales.
- **PyTorch:** Es una biblioteca de aprendizaje profundo de código abierto que es similar a TensorFlow, pero se centra en la velocidad y la eficiencia.
- **Keras:** Es un marco de aprendizaje profundo de alto nivel que se basa en TensorFlow o PyTorch.
- **SciPy:** Es una biblioteca de matemáticas y ciencias computacionales que proporciona funciones para el análisis de datos, la visualización y el cálculo numérico.
- **NumPy:** Es una biblioteca de cálculo numérico que proporciona matrices y funciones para el análisis de datos.
- **Pandas:** Es una biblioteca de análisis de datos que proporciona estructuras de datos y herramientas para la manipulación de datos tabulares.
- **Matplotlib:** Es una biblioteca de visualización de datos que proporciona funciones para crear gráficos y diagramas.

Estas bibliotecas proporcionan una amplia gama de funciones y capacidades para el aprendizaje computacional, incluyendo:

- **Algoritmos de aprendizaje automático:** La mayoría de estas bibliotecas proporcionan una amplia gama de algoritmos de aprendizaje automático, tanto para el aprendizaje supervisado como no supervisado.
- **Herramientas de preprocesamiento de datos:** Estas bibliotecas proporcionan herramientas para limpiar, transformar y preparar los datos para el aprendizaje automático.
- **Herramientas de evaluación de modelos:** Estas bibliotecas proporcionan herramientas para evaluar el rendimiento de los modelos de aprendizaje automático.
- **Herramientas de visualización de datos:** Estas bibliotecas proporcionan herramientas para visualizar los resultados del aprendizaje automático.

Elegir la biblioteca adecuada para una tarea de aprendizaje computacional depende de una serie de factores, incluyendo:

- **El tipo de algoritmo de aprendizaje automático que se necesita:** Algunas bibliotecas se especializan en ciertos tipos de algoritmos de aprendizaje automático, mientras que otras proporcionan una gama más amplia de opciones.

- **Las características y capacidades de la biblioteca:** Algunas bibliotecas proporcionan más funciones y capacidades que otras.
- **El nivel de experiencia del usuario:** Algunas bibliotecas son más fáciles de usar que otras.

En general, las bibliotecas de Python son una herramienta poderosa para el aprendizaje computacional. Proporcionan una amplia gama de funciones y capacidades que pueden ayudar a los usuarios a desarrollar modelos de aprendizaje automático eficaces.

25.6.3 Casos de estudio orientados al aprendizaje computacional

El aprendizaje computacional tiene una amplia gama de aplicaciones, y se utiliza en una gran variedad de industrias. Aquí hay algunos ejemplos de casos de estudio orientados al aprendizaje computacional:

- **Reconocimiento de imágenes y voz.** El reconocimiento de imágenes y voz es una de las aplicaciones más comunes del aprendizaje computacional. Se utiliza en una amplia gama de productos y servicios, como los asistentes virtuales, las cámaras de seguridad y los sistemas de reconocimiento facial.
- **Predicción.** El aprendizaje computacional se utiliza para predecir el comportamiento futuro. Se puede utilizar para predecir el riesgo de que un cliente abandone una empresa, la probabilidad de que un paciente desarrolle una enfermedad o el rendimiento de una inversión.
- **Robótica.** El aprendizaje computacional se utiliza para que los robots puedan aprender a realizar tareas en entornos complejos. Se puede utilizar para enseñar a los robots a navegar por un entorno, a interactuar con los humanos y a realizar tareas complejas.
- **Médico.** El aprendizaje computacional se utiliza en el campo de la medicina para diagnosticar enfermedades, personalizar tratamientos y desarrollar nuevos fármacos. Se puede utilizar para analizar imágenes médicas, para identificar patrones de enfermedad y para predecir la respuesta a los tratamientos.

Otros casos de estudio

El aprendizaje computacional se utiliza en una amplia gama de otros campos, como el marketing, la educación y el transporte. Por ejemplo, se puede utilizar para personalizar la publicidad, para mejorar el aprendizaje de los estudiantes y para optimizar el tráfico.

Ejemplos específicos

Aquí hay algunos ejemplos específicos de casos de estudio orientados al aprendizaje computacional:

- Google utiliza el aprendizaje computacional para clasificar imágenes en su motor de búsqueda.
- Amazon utiliza el aprendizaje computacional para recomendar productos a sus clientes.
- Netflix utiliza el aprendizaje computacional para recomendar películas y programas de televisión a sus usuarios.
- Tesla utiliza el aprendizaje computacional para que sus coches autónomos puedan navegar por la carretera.
- IBM Watson utiliza el aprendizaje computacional para diagnosticar enfermedades.
- Duolingo utiliza el aprendizaje computacional para personalizar el aprendizaje de idiomas.
- Uber utiliza el aprendizaje computacional para optimizar el tráfico.

Estos son solo algunos ejemplos de los muchos casos de estudio en los que el aprendizaje computacional se está utilizando para resolver problemas y mejorar nuestras vidas.

Capítulo 26

Pruebas de unidad

Capítulo 27

Multihilos: Introducción a la Programación Concurrente en Java

27.1 Introducción

Un programa concurrente es aquel que puede ejecutar varias tareas al mismo tiempo. Aunque en la vida diaria las cosas suceden concurrentemente o en paralelo, es interesante notar que los principales lenguajes de programación no permiten especificar actividades concurrentes de manera natural. Antes de Java, el lenguaje Ada implemento, como parte de su lenguaje, primitivas de concurrencia. Sin embargo, **Ada** no se hizo un lenguaje popular a pesar de haber sido el lenguaje oficial para el desarrollo de aplicaciones para el Departamento de Defensa de los Estados Unidos.

En general, si se quieren crear tareas concurrentes en un lenguaje como C++, se realiza a través de llamadas a primitivas de control del sistema operativo. Lógicamente, estas primitivas dependen del dominio que tenga el programador sobre la plataforma (no tanto del lenguaje) y varían de un sistema operativo a otro.

El lenguaje Java permite un manejo relativamente fácil de procesos concurrentes, respetando la independencia de la plataforma. Esto quiere decir que un programa con manejo de concurrencia en Java corre sin ninguna modificación en las máquinas cuyos sistemas operativos cuenten con una máquina virtual de java¹.

27.2 Concurrencia

Pero retrocedamos un poco y vamos a ponernos de acuerdo en el concepto de concurrencia. Veamos la diferencia entre concurrencia y paralelismo.

¹En realidad puede existir alguna diferencia en el comportamiento de los programas, pero se ahondará en el tema más adelante.

Creo que estaremos de acuerdo en que las **operaciones secuenciales** son aquellas que ocurren una después de otra; es decir, están ordenadas en el tiempo. De aquí se desprende que las **operaciones paralelas** son aquellas que ocurren al mismo tiempo. Es común hablar de paralelismo cuando hablamos de operaciones de hardware.

Sin embargo al hablar de concurrencia nos referimos por lo general al código fuente, donde un conjunto de operaciones son concurrentes si pueden ejecutarse en paralelo, lo que no quiere decir que obligatoriamente se ejecuten así. Es por eso que podemos tener procesos concurrentes sin tener hardware paralelo. Por ejemplo, las computadoras con *Windows* que realizan varios procesos al mismo tiempo como mandar a imprimir, bajar un archivo de Internet, enviar un *eMail*, etc. Las PC's son por lo general de un solo procesador y no cuentan con procesamiento paralelo; sin embargo, estamos realizando procesos concurrentes. Lo mismo sucede con todas las máquinas que tienen un solo procesador y cuentan con sistema operativo multitareas como *Unix* o *Linux*². Podemos decir que la concurrencia es cuando un conjunto de instrucciones no depende de otro conjunto y no importa el orden de ejecución entre ellas: son **potencialmente paralelas**.

27.3 Multihilos

Existen diversas técnicas de manejo de procesos concurrentes. Java maneja la concurrencia a través de los hilos (*threads*) de control. Los hilos tienen la característica de ejecutarse cada uno de ellos como un proceso independiente pero compartiendo un único espacio de direcciones. Estos procesos se conocen como **procesos ligeros** o hilos³, a diferencia de los demás procesos que no comparten el espacio de direcciones y donde la comunicación tiene que darse a través de primitivas de comunicación (semáforos, monitores o mensajes).

Los procesos ligeros o hilos son como miniprosesos, pues cada hilo se ejecuta de forma secuencial, con su propio contador de programa y pila de control. Además, al igual que los procesos, en una máquina de un solo procesador se van turnando su ejecución, lo que se conoce como **tiempo compartido**.

Anteriormente se mencionó que los hilos comparten un único espacio de direcciones. Esto quiere decir que tienen acceso a las mismas variables globales⁴ o ámbito de trabajo. En términos de objetos, los objetos de un hilo pueden ver a los de otro hilo si el alcance y las restricciones de acceso se lo permiten.

²Independientemente de que algunos sistemas operativos pueden ser usados en máquinas con múltiples procesadores.

³También llamados **contextos de ejecución**

⁴El concepto de variables globales no existe en Java, pero es un término común que nos da la idea del alcance. No olvidar además que el manejo de hilos nuevo o exclusivo del lenguaje Java.

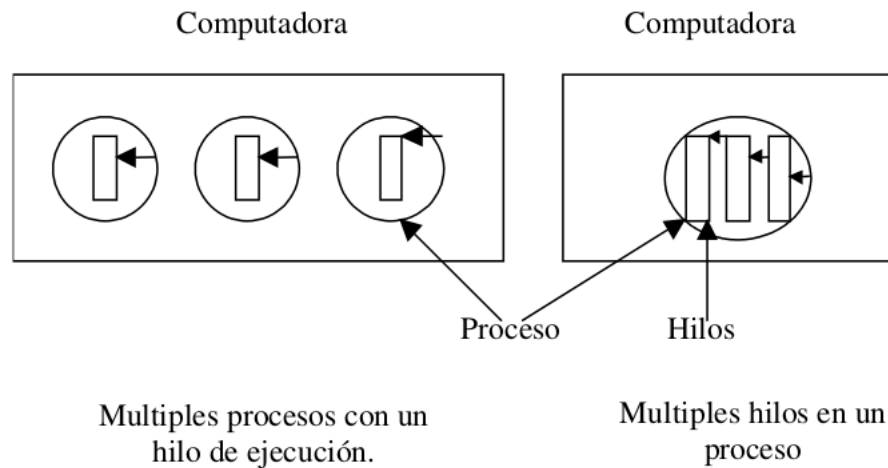


Fig. 27.1. Múltiples procesos vs. múltiples hilos en un proceso

27.4 Multihilos en Java

Aunque de manera estricta todos los programas de Java manejan más de un hilo, de vista al usuario los programas por lo general son de un único hilo de control (**flujo único**). Sin embargo pueden contar con varios hilos de control (**flujo múltiple**).

Existen dos formas de implementar hilos en un programa de Java. La forma más común es mediante herencia, extendiendo la clase *Thread*.

27.4.1 Programas de flujo único

Un programa de flujo único utiliza un único hilo de control para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchas de las aplicaciones son de flujo único.

Por ejemplo, en la aplicación clásica de "Hola mundo!":

```

1 public class HolaMundo {
2     static public void main( String args[] ) {
3         System.out.println( "Hola Mundo!" );
4     }
5 }
```

Aquí, cuando se llama a *main()*, la aplicación imprime el mensaje y termina. Esto ocurre dentro de un único hilo.

27.4.2 Programas de flujo múltiple

Los programas en Java implementan un flujo único de manera implícita. Sin embargo, Java posibilita la creación y control de hilos explícitamente. La utilización de hilos en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar hilos, permite que se puedan implementar muy poderosas y portables aplicaciones.

Las aplicaciones multihilos utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo para cada subtask.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multihilos permite que cada hilo comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a las necesidades de muchas aplicaciones.

Veamos un ejemplo de un pequeño programa multihilos en Java.

Ejemplo:

```
1  class MiHilo extends Thread {
2
3      public MiHilo (String nombre) {
4          super (nombre);
5      }
6
7      public void run() {
8          for( int i=0; i<4; i++){
9              System.out.println( getName() + " " + i );
10             try {
11                 sleep(400);
12             } catch( InterruptedException e) { }
13         }
14     }
15 }
16
17 public class MultiHilo {
18     public static void main(String arrg[]) {
19         MiHilo mascar = new MiHilo("Mascando");
20         MiHilo silbar = new MiHilo("Silbar");
21         mascar.start();
22         silbar.start();
23     }
24 }
```

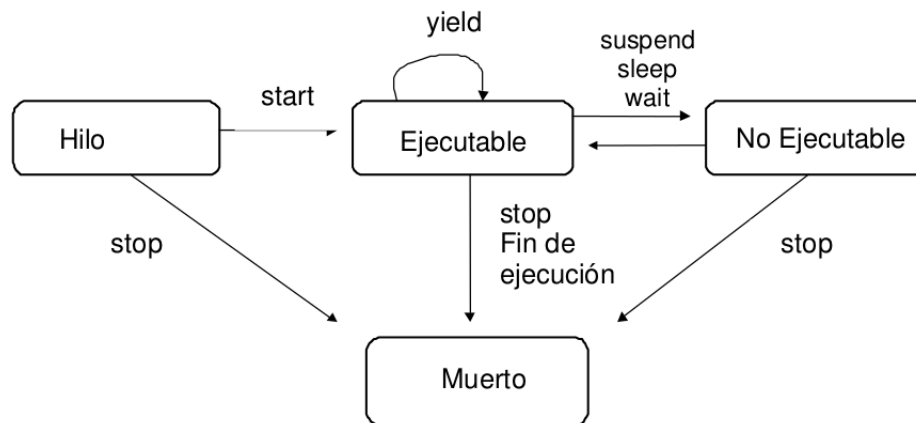


Fig. 27.2. Estados de un hilo en Java

Listing 241. Ejemplo básico de multihilos en Java.

Este pequeño ejemplo ejecuta dos hilos. Uno llamado *mascar* y otro *silbar*. Por lo que el programa es capaz de "mascar" y "silbar" al mismo tiempo, aunque como ya sabemos, en una computadora de un solo procesador tendrá que dejar de mascar para poder silbar, y viceversa.

27.5 Estados de un hilo

Cada hilo de ejecución en Java, es un objeto que puede estar en diferentes estados.

En la figura ?? podemos apreciar que cuando un hilo es creado, no quiere decir que se encuentre corriendo, sino que esto sucede cuando es invocado el método *start()* del objeto. Es hasta entonces que se encuentra en un estado de "Listo para ejecutarse".

Cuando un hilo está ejecutándose pueden pasar varias cosas con ese hilo en particular. El método *start()* llama en forma automática al método *run()*. Este método contiene el código principal del hilo, algo así como un método *main* para un programa principal.

Un hilo que está en ejecución puede pasar a un estado de muerto si termina de ejecutar al método *run()*.

El estado de "no-ejecutable". Se llega a este estado cuando el hilo no está "en ejecución", debido a una llamada del método *sleep()*, *wait()* o porque se está realizando un proceso de entrada/salida que tarda cierto tiempo en ejecutarse.

Existen los métodos *stop()*, *suspend()* y *resume()*, pero estos han sido desaprobadados en la versión 2 de Java debido a que se consideran potencialmente peligrosos

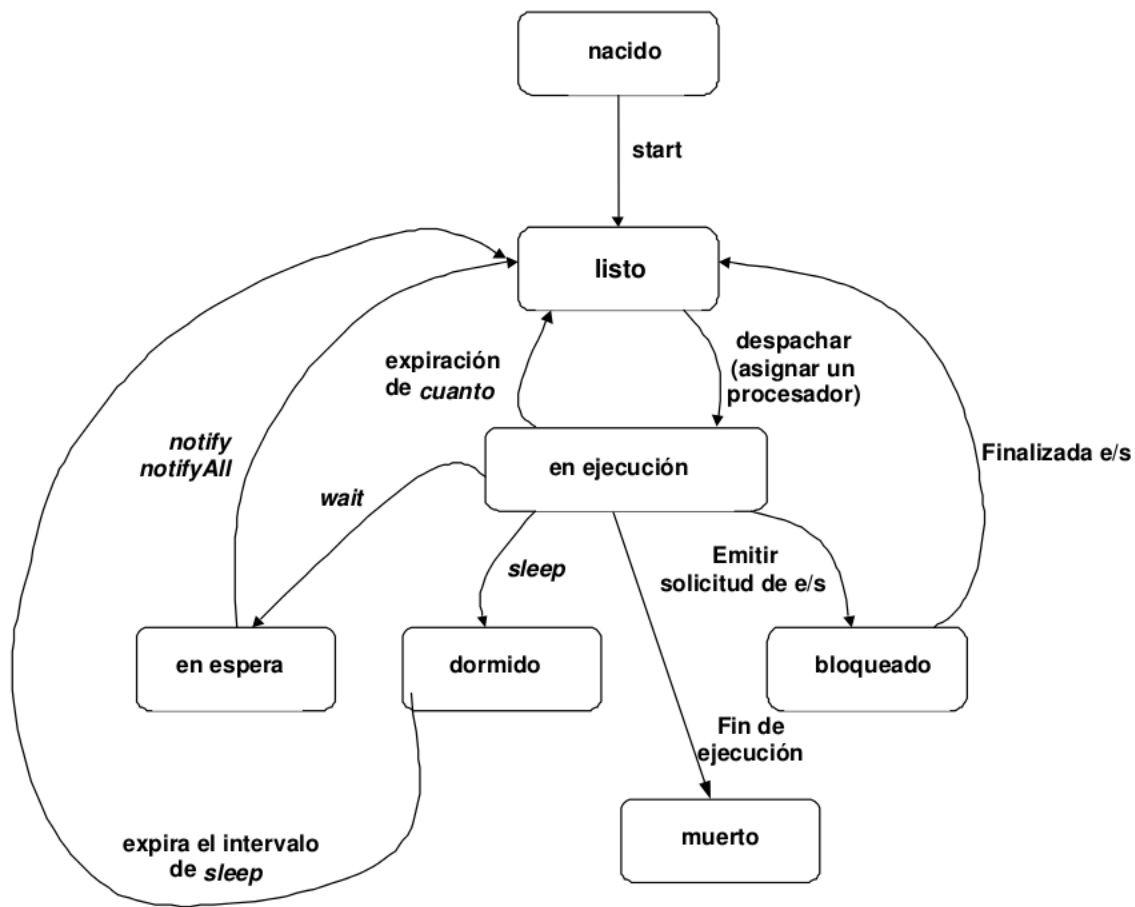


Fig. 27.3. Estados de un hilo en Java (2)

para la ejecución de los programas concurrentes⁵.

Veamos ahora un diagrama (figura ??) que muestra de manera más completa los estados en los que puede estar un hilo.

27.6 La clase *Thread*

El programa de ejemplo que se vio antes, corresponde a la forma de implementación más común de un hilo: mediante la extensión de la clase *Thread*. Por lo que se pudo apreciar, la sintaxis para la creación de un hilo sería:

⁵El que sean desaprobados no quiere decir que ya no puedan ser usados. Se conservan por compatibilidad hacia atrás con el lenguaje, pero se ha visto que no es recomendable su uso. En algunos ejemplos pueden aparecer estas instrucciones por simplicidad.

```
1 class MiHilo extends Thread {  
2     public void run() {  
3         . . .  
4     }  
5 }
```

Esta técnica, extiende a la clase *Thread*, y sobrescribe el método *run()*, el cual debe contener una implementación propia, de acuerdo a lo que se quiera que realice el hilo.

Vamos a mencionar ahora los principales métodos de la clase⁶.

- *Thread(String nombreThread)*. Constructor de la clase *Thread*, recibe una cadena para el nombre del hilo.
- *Thread()*. Constructor sin parámetros. Crea de manera automática nombres para los hilos. Llamados *Thread1*, *Thread2*, etc.
- *start()*. Inicia la ejecución de un hilo. Invoca al método *run()*.
- *run()*. Este método se sobrescribe para controlar la ejecución del hilo.
- *sleep(tiempo)*. Causa que el hilo se "duerma" un tiempo determinado. Un hilo dormido no compete por el procesador.
- *interrupt()*. Interrumpe la ejecución de un hilo.
- *interrupted()*. Método estático que devuelve verdadero si el hilo actual ha sido interrumpido.
- *isInterrupted()*. Método no estático que verifica si un hilo ha sido interrumpido.
- *join()*. Espera a que un hilo específico muera antes de continuar. Está sobrecargado para recibir un tiempo límite de espera como parámetro.
- *yield()*. El hilo cede la ejecución a otros hilos.

Métodos usados también con los hilos son *notify()* y *wait()*, aunque estos en realidad están definidos y heredados desde la clase *Object*.

⁶Para las características completas ver la documentación: *Java Platform API Specification*

27.7 Prioridades de los hilos

A cada uno de los hilos de Java se les puede asignar una **prioridad**. ¿Por qué es esto necesario? Recordemos que en las máquinas con un solo procesador solo se tiene la impresión de que todos los hilos se están ejecutando al mismo tiempo, cuando en realidad se están repartiendo el tiempo del procesador.

Ahora, suponga que tiene un programa con varios hilos de ejecución, y uno de los hilos lleva a cabo una tarea más importante o que requiera mayores recursos. Una opción lógica sería asignarle una prioridad mayor que al resto de los hilos para que tenga preferencia al competir por tiempo del procesador.

La prioridad de un hilo en Java puede ir de 1 a 10. Si se le asigna 10 como prioridad a un hilo, se le estaría asignando la prioridad más alta. Un hilo al que no se le especifica su prioridad, toma una prioridad normal con un valor de 5. Si un hilo es creado por otro, el nuevo hilo hereda la prioridad del hilo padre.

Java tiene psobreescritas 3 constantes de prioridad:

- *MIN_PRIORITY*. Para asignar la mínima prioridad que puede tener un hilo (1).
- *NORM_PRIORITY*. Prioridad intermedia (5), asignada por omisión.
- *MAX_PRIORITY*. Asigna la máxima prioridad que puede tener un hilo (10).

El siguiente ejemplo muestra el uso de prioridades en los hilos e introduce además el manejo del método *setPriority*; el cual, junto con *getPriority*, sirven para modificar y consultar la prioridad de un hilo respectivamente.

Ejemplo:

```
1  class miHilo extends Thread {
2
3      public miHilo (String nombre)
4      {
5          super (nombre);
6      }
7
8      public void run()
9      {
10         for (int i = 0; i < 20; i++)
11         {
12             System.out.println(getName() + " " + i);
13             try {
14                 sleep(10);
15             }
```

```
16         catch (InterruptedException e) {}
17     }
18 }
19 }
20
21 public class prioridadHilo {
22
23
24     public static void main(String args[])
25     {
26         miHilo hilo_min = new miHilo ("Hilo Min");
27         miHilo hilo_max = new miHilo ("Hilo Max");
28
29         hilo_min.setPriority(Thread.MIN_PRIORITY);
30         hilo_max.setPriority(Thread.MAX_PRIORITY);
31         hilo_min.start();
32         hilo_max.start();
33     }
34 }
```

Listing 242. Ejemplo de prioridades en multihilos con Java.

Mostramos a continuación un ejemplo parecido que genera cuatro hilos y son puestos a dormir un tiempo aleatorio, de entre 0 y 5 segundos. En este caso todos los hilos tienen la misma prioridad, y su ejecución depende del momento en que soliciten el procesador.

Ejemplo:

```
1 // Muestra múltiples hilos desplegándose a diferentes
2 // intervalos.
3
4 public class PrintTest {
5     public static void main( String args[] )
6     {
7         PrintThread thread1, thread2, thread3, thread4;
8
9         thread1 = new PrintThread( "1" );
10        thread2 = new PrintThread( "2" );
11        thread3 = new PrintThread( "3" );
12        thread4 = new PrintThread( "4" );
13
14        thread1.start();
```

```
15     thread2.start();
16     thread3.start();
17     thread4.start();
18 }
19 }
20
21 class PrintThread extends Thread {
22     int sleepTime;
23
24     // constructor PrintThread asigna nombre al hilo
25     // llamando al constructor de Thread
26     public PrintThread( String id )
27     {
28         super( id );
29
30         // valor aleatorio para dormir el hilo de 0 a 5 segundos
31         sleepTime = (int) ( Math.random() * 5000 );
32
33         System.out.println( "Nombre: " + getName() +
34                             "; durmiendo: " + sleepTime );
35     }
36
37     // ejecuta el hilo
38     public void run()
39     {
40         try {
41             sleep( sleepTime );
42         }
43         catch ( InterruptedException exception ) {
44             System.err.println( "Excepcion: " +
45                                 exception.toString() );
46         }
47         System.out.println( "Hilo " + getName() );
48     }
49 }
```

Listing 243. Ejemplo múltiples hilos desplegándose a diferentes // intervalos.

27.8 Comportamiento de los hilos

La implementación real de los hilos puede variar un poco de una plataforma a otra. Algunos sistemas como *Windows*, los hilos funcionan por **rebanadas de tiempo** y otros como muchas versiones de *Unix* no tienen esta característica.

En los sistemas que se manejan rebanadas de tiempo, los hilos de igual prioridad se reparten el tiempo de ejecución en partes iguales. En los sistemas que no tienen rebanadas de tiempo, un hilo se ejecuta hasta que cede el control voluntariamente, se lo quita un hilo de **mayor** prioridad, o termina su ejecución.

Bajo este último esquema, es importante que un hilo delegue el control cada determinado tiempo a hilos de igual prioridad. Para esto sirve poner a dormir el hilo con *sleep()*, o ceder el control con el método *yield()*. Un método que tiene estas consideraciones se conoce como hilo compartido, el caso contrario se conoce como **hilo egoísta**.

Tener en cuenta que el método *yield()* cede el control a hilos de la misma prioridad. Esto es útil en plataformas que no cuenten con rebanadas de tiempo, pero no tiene sentido en sistemas que si cuentan con esta técnica⁷.

Veamos una clase que implementa un hilo y cede el control a otros hilos.

Ejemplo:

```
1  class HiloEterno extends Thread {
2
3      public HiloEterno (String nombre) {
4          super (nombre);
5      }
6
7      public void run()
8      {
9          int i=0;
10
11         while (true)      // Iterar para siempre
12         {
13             System.out.println(getName() + " " + "Ciclo " + i++);
14             if (i%100==0)
15                 yield();    // Ceder el procesador a otros hilos
16         }
17     }
18 }
19
```

⁷Sin embargo debería siempre considerarse el uso de *yield()* si se piensa en sistemas multiplataformas.

```
20 public class MultiHilo2 {  
21     public static void main(String arg[]) {  
22         HiloEterno infinito = new HiloEterno("Al infinito");  
23         HiloEterno masAlla = new HiloEterno("y mas alla");  
24         infinito.start();  
25         masAlla.start();  
26     }  
27 }  
28
```

Listing 244. Ejemplo que implementa un hilo y cede el control a otros hilos.

Capítulo 28

Multihilos: Programación Concurrente en Java (2)

28.1 Sincronización de hilos

Es lógico pensar que un programa concurrente puede tener problemas al tratar de acceder datos comunes. Por ejemplo, si dos hilos tratan de manipular una variable al mismo tiempo, puede ocasionarse un valor inesperado en la misma. Para evitar este tipo de problemas, Java utiliza **monitores**¹, permitiendo poner un **candado** para que un solo hilo pueda acceder a los datos a la vez.

Existen dos formas de proteger el acceso a datos. El primero es ejecutar un método sincronizado, el cual es definido mediante la palabra reservada *synchronized*, de manera que **sólo un método sincronizado puede ejecutarse para el objeto**. Hasta que termina de ejecutarse un método sincronizado, se permite que entre a ejecutarse el hilo con mayor prioridad, el cual puede ser a su vez un método sincronizado².

Veamos un primer ejemplo de sincronización. Se trata del control de una cuenta de cheques, en donde es posible que se trate de cobrar más de un cheque al mismo tiempo (en este caso se muestran dos hilos). El problema se daría si el método que modifica el balance de la cuenta no estuviera sincronizado, pues pudiera darse el caso de que no se registren correctamente los cambios. La modificación del balance es una **sección crítica**, y por lo mismo se encuentra protegida con monitores.

[Ejemplo:](#)

¹Un monitor es una colección de variables y procedimientos compartidos, con la restricción de que sólo un proceso a la vez puede ejecutar un procedimiento del monitor.

²Una opción adicional de implementar sincronización en Java es mediante la clase *ReentrantLock*, pero no se verá en el curso. Ver: [Reentrant lock Java](#)

```
1 class Cuenta {
2
3     static int balance = 1000;
4     static int gasto = 0;
5
6     static public synchronized void retirar(int cantidad)
7     {
8         if (cantidad <= balance)
9         {
10             System.out.println("cheque: " + cantidad);
11             balance -= cantidad;
12             gasto += cantidad;
13             System.out.print("balance: " + balance);
14             System.out.println(", se gastó: " + gasto);
15         }
16         else
17         {
18             System.out.println("rebotó: " + cantidad);
19         }
20     }
21 }
22
23 class miHilo extends Thread {
24     public void run()
25     {
26         for (int i = 0; i < 10; i++)
27         {
28             try {
29                 sleep(100);
30             }
31             catch (InterruptedException e) {}
32             Cuenta.retirar((int) (Math.random() * 500 ));
33         }
34     }
35 }
36
37 public class Sincronizar {
38
39     public static void main(String arg[]){
40         new miHilo().start();
41         new miHilo().start();
```

```
42     }  
43 }  
44
```

Listing 245. Ejemplo de sincronización de hilos.

Además de declarar un método como *synchronized*, es posible declarar a un objeto y un bloque de código como sincronizados. Esto determina que **sólo un hilo puede ejecutar métodos del objeto en cuestión**. Por lo tanto, nadie puede modificar sus atributos más que el bloque de código declarado como sincronizado. Veamos el mismo ejemplo anterior implementado ahora con esta técnica:

Ejemplo:

```
1  class miHilo extends Thread {  
2  
3      static Integer balance = new Integer(1000);  
4      static int gasto = 0;  
5      static { // bloque estático se ejecuta cuando la JVM carga la clase  
6          System.out.print("Balance inicial: " + balance);  
7      }  
8  
9      public void run()  
10     {  
11         int cuenta;  
12  
13         for (int i = 0; i < 10; i++)  
14         {  
15             try {  
16                 sleep(100);  
17             }  
18             catch (InterruptedException e) {}  
19  
20             cuenta = ((int) (Math.random() * 500 ));  
21             synchronized (balance)  
22             {  
23                 if (cuenta <= balance.intValue())  
24                 {  
25                     System.out.println("cheque: " + cuenta);  
26                     balance = new Integer(balance.intValue()-cuenta);  
27                     gasto += cuenta;  
28                     System.out.print("bal: "+balance.intValue());  
29                     System.out.println(", se gastó: " + gasto);  
30                 } else
```

```
31         {
32             System.out.println("rebotó: " + cuenta);
33         }
34     }
35 }
36 }
37 }
38
39 public class ObjetoSincronizado {
40
41     public static void main(String args[]) {
42         new miHilo().start();
43         new miHilo().start();
44     }
45 }
```

Listing 246. Ejemplo de multihilos declarando un bloque de código sincronizado.

28.1.1 Problema Productor / Consumidor

En el sentido clásico, el problema del productor /consumidor se presenta cuando en datos compartidos, un proceso necesita cierto dato (**consumidor**) que está preparando otro proceso (**productor**). Es lógico que ninguno de los dos deben acceder el dato al mismo tiempo. Pero también puede suceder que el consumidor llegue antes de que el productor tenga listos los datos; o viceversa, que el productor genere los datos y el consumidor todavía no pueda tomarlos.

El siguiente ejemplo muestra un caso de Productor /Consumidor **sin** sincronización. Existe un objeto productor que va generando números consecutivos y un objeto consumidor que los va obteniendo. En la ejecución se podrá apreciar que el consumidor ocasionalmente recupera un número más de una vez, lo cual no debería ocurrir³.

Ejemplo:

```
1 public class SharedCell {
2     public static void main( String args[] )
3     {
4         HoldInteger h = new HoldInteger();
5         ProduceInteger p = new ProduceInteger( h );
```

³Por la forma en que está programado el objeto productor, éste no genera dos veces el mismo número. Pero se daría este caso si obtuviera el último número generado del valor compartido como base para generar el siguiente número.

```
6      ConsumeInteger c = new ConsumeInteger( h );
7
8      p.start();
9      c.start();
10     }
11 }
12
13 class ProduceInteger extends Thread {
14     private HoldInteger pHold;
15
16     public ProduceInteger( HoldInteger h )
17     {
18         pHold = h;
19     }
20
21     public void run()
22     {
23         for ( int count = 0; count < 10; count++ ) {
24             pHold.setSharedInt( count );
25             System.out.println( "El productor puso: " +
26                               count );
27
28             try {
29                 sleep( (int) ( Math.random() * 3000 ) );
30             }
31             catch( InterruptedException e ) {
32                 System.err.println( "Excepcion " + e.toString());
33             }
34         }
35     }
36 }
37
38 class ConsumeInteger extends Thread {
39     private HoldInteger cHold;
40
41     public ConsumeInteger( HoldInteger h )
42     {
43         cHold = h;
44     }
45
46     public void run()
47     {
```

```
48     int val;
49
50     val = cHold.getSharedInt();
51     System.out.println( "Recuperado por el consumidor: " + val );
52
53     while ( val != 9 ) {
54
55         try {
56             sleep( (int) ( Math.random() * 3000 ) );
57         }
58         catch( InterruptedException e ) {
59             System.err.println( "Excepcion " + e.toString() );
60         }
61
62         val = cHold.getSharedInt();
63         System.out.println( "Recuperado por el consumidor: " + val );
64     }
65 }
66
67
68 class HoldInteger {
69     private int sharedInt;
70
71     public void setSharedInt( int val ) {
72         sharedInt = val;
73     }
74
75     public int getSharedInt() {
76         return sharedInt;
77     }
78 }
```

Listing 247. Ejemplo: Problema Productor / Consumidor. Modificación de un objeto compartido sin sincronización.

A continuación veremos el mismo programa, pero resuelto con sincronización:

Ejemplo:

```
1  //Problema Productor /Consumidor
2  // con sincronizacion de hilos.
3
4  public class SharedCell {
5      public static void main( String args[] )
```



```
6      {
7          HoldInteger h = new HoldInteger();
8          ProduceInteger p = new ProduceInteger( h );
9          ConsumeInteger c = new ConsumeInteger( h );
10
11          p.start();
12          c.start();
13      }
14  }
15
16  class ProduceInteger extends Thread {
17      private HoldInteger pHold;
18
19      public ProduceInteger( HoldInteger h )
20      {
21          pHold = h;
22      }
23
24      public void run()
25      {
26          for ( int count = 0; count < 10; count++ ) {
27              pHold.setSharedInt( count );
28              System.out.println( "Productor asigna a sharedInt el valor: " +
29                               count );
30
31              try {
32                  sleep( (int) ( Math.random() * 3000 ) );
33              }
34              catch( InterruptedException e ) {
35                  System.err.println( "Excepcion " + e.toString() );
36              }
37          }
38      }
39  }
40
41  class ConsumeInteger extends Thread {
42      private HoldInteger cHold;
43
44      public ConsumeInteger( HoldInteger h )
45      {
46          cHold = h;
47      }
```

```
48
49     public void run()
50     {
51         int val;
52
53         val = cHold.getSharedInt();
54         System.out.println( "Recuperado por Consumidor: " + val );
55
56         while ( val != 9 ) {
57             try {
58                 sleep( (int) ( Math.random() * 3000 ) );
59             }
60             catch( InterruptedException e ) {
61                 System.err.println( "Excepcion " + e.toString() );
62             }
63
64             val = cHold.getSharedInt();
65             System.out.println( "Recuperado por consumidor: " + val );
66         }
67     }
68 }
69
70 class HoldInteger {
71     private int sharedInt;
72     private boolean writeable = true;
73
74     public synchronized void setSharedInt( int val )
75     {
76         while ( !writeable ) {
77             try {
78                 wait();
79             }
80             catch ( InterruptedException e ) {
81                 System.err.println( "Excepcion: " + e.toString() );
82             }
83         }
84
85         sharedInt = val;
86         writeable = false;
87         notify();
88     }
89 }
```

```
90     public synchronized int getSharedInt ()
91     {
92         while ( writeable ) {
93             try {
94                 wait();
95             }
96             catch ( InterruptedException e ) {
97                 System.err.println( "Excepcion: " + e.toString() );
98             }
99         }
100
101         writeable = true;
102         notify();
103         return sharedInt;
104     }
105 }
```

Listing 248. Ejemplo: Problema Productor /Consumidor // con sincronizacion de hilos.

En el ejemplo anterior se introducen los métodos *wait()* y *notify()*. Estos ayudan a mejorar el funcionamiento de los métodos sincronizados. Cuando un hilo sincronizado no puede continuar por algún motivo, deberá llamar al método *wait()*, permitiendo que el hilo deje de competir por el tiempo de procesador y que otro hilo pueda procesar los datos compartidos. El método *notify()* es ocupado para avisar a un hilo que se encuentra en espera, que el hilo que hizo la llamada ha terminado de realizar su proceso crítico y que pueden intentar ejecutarse, obteniendo para esto el **candado** del objeto monitor.

Veamos un último ejemplo para dejar claro el uso de *wait()* y *notify()*, usando el mismo concepto de Productor/consumidor.

Ejemplo:

```
1  class Tienda {
2      int mercancia = 0;
3
4      public synchronized int consumir()
5      {
6          int temp;
7          while (mercancia == 0)
8              {
9                  try {
10                     wait();
11                 }
```

```
12         catch (InterruptedException e) {}
13     }
14
15     temp = mercancia;
16     mercancia = 0;
17     System.out.println("Se consumió: " + temp);
18     notify();
19     return temp;
20 }
21
22 public synchronized void producir(int cantidad)
23 {
24     while (mercancia != 0)
25     {
26         try {
27             wait();
28         }
29         catch (InterruptedException e) {}
30     }
31
32     mercancia = cantidad;
33     notify();
34     System.out.println("Se produjo: " + mercancia);
35 }
36 }
37
38 class miHilo extends Thread {
39
40     boolean productor = false;
41     Tienda tienda;
42
43     public miHilo(Tienda d, String tipo)
44     {
45         tienda = d;
46
47         if (tipo.equals("Productor"))
48             productor = true;
49     }
50
51     public void run()
52     {
53         for (int i = 0; i < 10; i++)
```

```
54     {
55         try {
56             sleep((int) (Math.random() * 200 ));
57         }
58         catch (InterruptedException e) {}
59
60         if (productor)
61             tienda.producir((int) (Math.random() * 10 ) + 1);
62         else // debe ser un consumidor
63             tienda.consumir();
64     }
65 }
66
67
68 public class WaitNotify {
69
70     Tienda tienda = new Tienda();
71
72     public static void main(String args[]){
73         WaitNotify wn = new WaitNotify();
74         wn.ini();
75     }
76
77     public void ini() {
78         new miHilo(tienda, "Consumidor").start();
79         new miHilo(tienda, "Productor").start();
80     }
81 }
```

Listing 249. Ejemplo adicional de productor/consumidor.

Esperando que haya quedado claro el uso de *wait()* y *notify()*, mencionaremos algunas observaciones que consideramos importantes. En primer lugar, un hilo que es puesto en espera mediante el método *wait()*, debe ser notificado en algún momento de que puede continuar, de lo contrario puede quedarse esperando indefinidamente. Es vital entonces que para cada llamada a *wait()* haya una correspondiente llamada a *notify()*. Un hilo también puede ser mandado a la cola de espera porque trate de entrar un método sincronizado y ya se encuentre en ejecución un método sincronizado para ese objeto; pero, a diferencia de un hilo puesto en espera explícitamente, este hilo se reactivará cuando tenga oportunidad de entrar sin necesidad de que le notifiquen.

El método *notify()* sólo le indica a un hilo que puede dejar de esperar. Existe además un método *notifyAll()*, que hace que todos los hilos que están en espera

traten de ejecutarse. Obviamente sólo uno podrá hacerlo, pero todos tendrán la oportunidad de entrar.

Hay que tener cuidado con las sincronizaciones, ya que demasiado código sincronizado puede hacer muy lenta la ejecución de los hilos. Se debe sincronizar únicamente cuando se trate de situaciones críticas que tienen que ver con datos compartidos.

Un problema común es que un programa muy sincronizado genere una ejecución casi secuencial. Otro problema más grave es que puede llegarse a dar el caso de que todos los hilos se queden esperando, esto se conoce como **abrazo mortal** (*deadlock*), y por supuesto que hay que tratar de evitarlo⁴. El abrazo mortal se puede evitar con una cuidadosa programación que promueva el uso ordenado de los recursos⁵.

28.2 Grupos de hilos

Cuando se tiene un programa que necesita manejar múltiples hilos, es posible que varios de ellos estén trabajando en procesos similares. En Java, es posible agrupar los hilos para poder manipularlos de manera más eficiente, ya que se pueden enviar mensajes al grupo de hilos - como una unidad - en lugar de tener que hacerlo a cada uno de los hilos.

Para lograr esto, se provee de la clase *ThreadGroup*, la cual proporciona métodos para el control de los hilos que en general son versiones de grupo de los métodos que proporciona la clase *Thread*. La formación de un grupo de hilos puede darse de la siguiente forma:

```
1 ThreadGroup nombreGrupo = new ThreadGroup( "nombre del grupo");
2 miHilo mh1 = new miHilo (nombreGrupo, "hilo 1");
3 miHilo mh2 = new miHilo (nombreGrupo, "hilo 2");
```

28.3 Hilos Demonios

Es posible tener en Java hilos demonio o *daemon*. Estos hilos tienen este nombre ya que -como los *daemon* de un sistema operativo- son procesos servidores que se encuentran corriendo con el propósito de proporcionar un servicio a otros hilos.

Estos hilos *daemon* se ejecutan por lo general con una prioridad baja, y si no hay ningún otro hilo al que le deban prestar servicio, finalizan. Un ejemplo de un hilo *daemon* es el recolector de basura⁶ de Java. Para que un hilo definido por

⁴Investigar sobre el problema de los filósofos tragones.

⁵Existen algunos métodos formales que salen del alcance de este curso.

⁶garbage collector

nosotros sea considerado *daemon* debe especificarse mediante el uso del método *setDaemon()*:

```
1 Thread miHilo= new Thread();
2 miHilo.setDaemon(true);
```

Es importante señalar que un hilo debe de ser determinado como *daemon* antes de que se ejecute el método *start()* de ese hilo, o será lanzada la excepción *IllegalThreadStateException*.

28.4 La interfaz *Runnable*

Se ha visto la implementación de hilos heredando de la clase *Thread*, la cual es la forma más común. Pero puede ser que la clase, a la que queremos implementar funcionalidades de hilos, ya tenga definida herencia de otra clase. Como Java no soporta herencia múltiple, para implementar el hilo en la clase tendría que hacerse a través de la interfaz *Runnable*.

Esta interfaz únicamente tiene definido el método *run()*, el cual debe implementarse al igual que en la clase que se extiende la clase *Thread*. El esquema general para usar la interfaz *Runnable* es:

```
1 public class MiHilo implements Runnable {
2
3     public void run() {
4         //código del hilo
5     }
6 }
```

El inicio de la ejecución de un hilo difiere cuando se implementa esta interfaz, ya que se debe crear una instancia de la clase *Thread* y pasarle como parámetro al constructor del hilo un objeto de la clase que implementa la interfaz *Runnable*:

```
1 Thread miHilo= new Thread( new MiHilo() );
2
3 miHilo.start();
```

Ejemplo:

```
1 public class DemoThread implements Runnable {
2
3     @Override
4     public void run() {
5         for (int i = 0; i < 3; i++) {
6             System.out.println("Hilo Hijo ");
7
8             try {
9                 Thread.sleep(200);
10            } catch (InterruptedException ie) {
11                System.out.println("Hilo hijo se ha interrumpido! " + ie);
12            }
13        }
14
15        System.out.println("Hilo hijo finalizado!");
16    }
17
18    public static void main(String[] args) {
19        Thread t = new Thread(new DemoThread ());
20
21        t.start();
22
23        for (int i = 0; i < 3; i++) {
24            System.out.println("Hilo principal");
25            try {
26                Thread.sleep(200);
27            } catch (InterruptedException ie) {
28                System.out.println("Hilo hijo interrumpido! " + ie);
29            }
30        }
31        System.out.println("Hilo principal finalizado!");
32    }
33 }
```

Listing 250. Ejemplo de multihilos usando la interfaz Runnable.

Ejemplo:⁷

```
1 // Ejemplo de implementacion de la interfaz Runnable con JavaFX
2 import javafx.application.Application;
```

⁷Este ejemplo ha sido actualizado para usar JavaFX y mecanismos modernos de sincronización, reemplazando los métodos obsoletos suspend() y resume().


```
3 import javafx.application.Platform;
4 import javafx.stage.Stage;
5 import javafx.scene.Scene;
6 import javafx.scene.control.Button;
7 import javafx.scene.control.TextField;
8 import javafx.scene.layout.VBox;
9 import javafx.scene.layout.HBox;
10 import javafx.event.ActionEvent;
11 import javafx.event.EventHandler;
12
13 public class RandomCharacters extends Application {
14
15     private TextField output1, output2, output3;
16     private Button button1, button2, button3;
17     private CharacterRunner runner1, runner2, runner3;
18     private Thread thread1, thread2, thread3;
19
20     @Override
21     public void start(Stage primaryStage) {
22         VBox root = new VBox(10);
23
24         output1 = new TextField(); output1.setEditable(false);
25         output2 = new TextField(); output2.setEditable(false);
26         output3 = new TextField(); output3.setEditable(false);
27
28         button1 = new Button("Suspender/Continuar 1");
29         button2 = new Button("Suspender/Continuar 2");
30         button3 = new Button("Suspender/Continuar 3");
31
32         // Configurar acciones de botones
33         button1.setOnAction(e -> toggleThread(runner1));
34         button2.setOnAction(e -> toggleThread(runner2));
35         button3.setOnAction(e -> toggleThread(runner3));
36
37         root.getChildren().addAll(output1, button1, output2, button2, output3, button3);
38
39         // Iniciar hilos
40         runner1 = new CharacterRunner(output1, "Hilo 1");
41         runner2 = new CharacterRunner(output2, "Hilo 2");
42         runner3 = new CharacterRunner(output3, "Hilo 3");
43
44         thread1 = new Thread(runner1);
```

```
45     thread2 = new Thread(runner2);
46     thread3 = new Thread(runner3);
47
48     thread1.setDaemon(true);
49     thread2.setDaemon(true);
50     thread3.setDaemon(true);
51
52     thread1.start();
53     thread2.start();
54     thread3.start();
55
56     Scene scene = new Scene(root, 300, 400);
57     primaryStage.setTitle("Hilos y JavaFX");
58     primaryStage.setScene(scene);
59     primaryStage.show();
60 }
61
62 private void toggleThread(CharacterRunner runner) {
63     runner.toggle();
64 }
65
66 public static void main(String[] args) {
67     launch(args);
68 }
69
70 // Clase interna que implementa Runnable
71 class CharacterRunner implements Runnable {
72     private TextField output;
73     private String name;
74     private volatile boolean suspended = false;
75     private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
76
77     public CharacterRunner(TextField output, String name) {
78         this.output = output;
79         this.name = name;
80     }
81
82     public synchronized void toggle() {
83         suspended = !suspended;
84         if (!suspended) {
85             notify();
86         }
87     }
88 }
```

```
87     }
88
89     @Override
90     public void run() {
91         while (true) {
92             try {
93                 synchronized (this) {
94                     while (suspended) {
95                         wait();
96                     }
97                 }
98
99                 int location = (int) (Math.random() * 26);
100                 char display = alphabet.charAt(location);
101
102                 // Actualizar UI en el hilo de JavaFX
103                 Platform.runLater(() -> output.setText(name + ": " + display));
104
105                 Thread.sleep((int) (Math.random() * 1000));
106             } catch (InterruptedException e) {
107                 break;
108             }
109         }
110     }
111 }
112 }
```

Listing 251. Ejemplo de implementación de la interfaz *Runnable* con una aplicación JavaFX.

28.5 Actividades sugeridas

Hacer un programa que simule el problema de los filósofos tragones (ver figura ??).

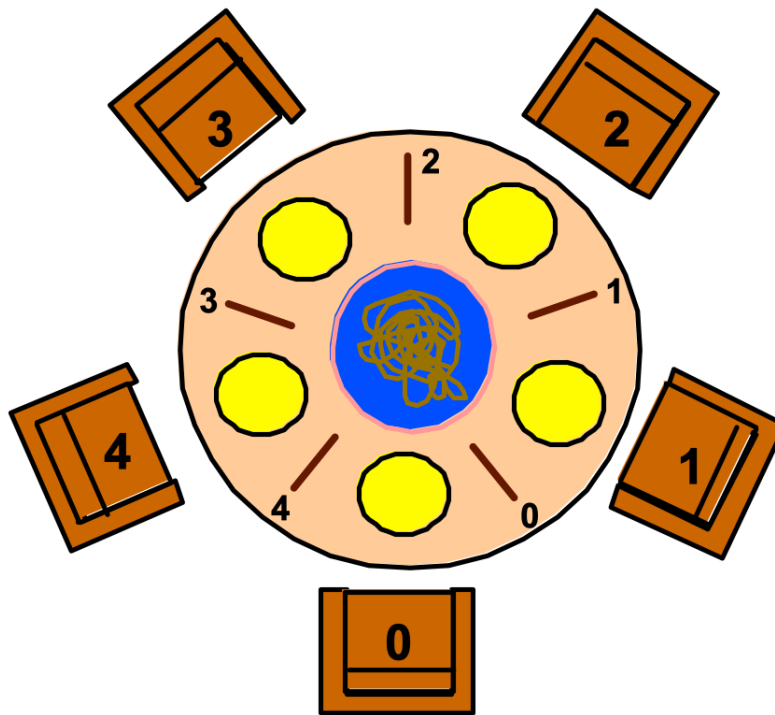


Fig. 28.1. Filósofos tragones

Lecturas complementarias recomendadas

- Writing multithreaded Java applications⁸
- Synchronization is not the enemy
- Reducing contention
- Threading lightly : Sometimes it's best not to share

Capítulo 29

Multihilos en Python

29.1 Forking (Bifurcación)

Una manera relativamente simple de implementar procesos concurrentes en Python es mediante *forking* o bifurcación de procesos.

La bifurcación de procesos son una forma tradicional de ejecutar tareas concurrentes en Unix. La bifurcación esta basada en la idea de *copia de programas*. Cuando un programa llama a una rutina bifurcada, el sistema operativo realiza una nueva copia del programa y su proceso en memoria e inicia la ejecución de la copia en paralelo con el programa original.

El programa original es llamado *proceso padre*, mientras la copia creada es llamada *proceso hijo*.

El soporte de Python para bifurcación se encuentra en el módulo *os* y son en realidad llamadas encapsuladas a las operaciones del sistema para manejo de procesos.

Para iniciar un nuevo proceso concurrente se llama a la función *os.fork()*, lo que genera una copia del programa. Esta llamada regresa el valor de cero en el proceso hijo y el ID del proceso del nuevo hijo en el proceso padre. Este valor de cero generalmente es usado para diferenciar el código de ejecución del proceso hijo.

La bifurcación es parte del modelo Unix por lo que funciona bien en sistemas Unix, Linux y OS X, pero no funciona adecuadamente en Windows.

Veamos un ejemplo simple de bifurcación de procesos.

```
1  #Bifurca procesos hijos hastq eu se introduce 's'
2
3  import os
4
5  def hijo():
```

```

6     print('Hola desde el proceso hijo', os.getpid())
7     os._exit(0) # si no regresa al ciclo padre
8
9     def padre():
10         while True:
11             newpid = os.fork()
12             if newpid == 0:
13                 hijo()
14             else:
15                 print('Hola desde el proceso padre', os.getpid(), newpid)
16                 if input("Salir(s)") == 's':
17                     break
18
19     padre()

```

Listing 252. Ejemplo de bifurcación de procesos (Forking).

El siguiente ejemplo iniciará cinco copias y sale inmediatamente. Cada copia tiene un retardo de un segundo y se ejecuta a pesar de que el proceso padre ha terminado.

```

1     """
2     Inicia 5 copias del programa corriendo en paralelo con el original;
3     cada copia cuenta hasta 5 en el mismo flujo de salida estándar,
4     bifurca copiando la memoria del proceso.
5     No trabaja en Windows sin Cygwin
6     """
7
8     import os, time
9
10    def contador(count): #ejecuta un nuevo proceso
11        for i in range(count):
12            time.sleep(1) # simula que trabaja
13            print(f'[{os.getpid()}] => {i}')
14
15    for i in range(5):
16        pid = os.fork()
17        if pid != 0:
18            print(f'Proceso {pid} generado') # en proceso padre: continua
19        else:
20            contador(5) # sino in proceso hijo/nuevo
21            os._exit(0) # ejecuta función y sale

```

```
22
23 print('Saliendo del proceso principal.') #proceso padre no necesita esperar
24
```

Listing 253. Ejemplo de bifurcación de 5 procesos (Forking).

29.1.1 fork/exec

La combinación de *fork* con ejecución de programas en sistemas operativos tipo Unix son otra forma simple de iniciar procesos concurrentes. Esto implica por un lado la creación de un proceso con la llamada al sistema operativo de ejecutar otro programa, para lo cual se puede ocupar alguna de los diferentes formatos de llamadas a funciones *os.exec*. Por ejemplo, una llamada a la función *os.execlp()*, le especifica al programa que se ejecute a partir de la línea de comandos usada para iniciar el programa. El nuevo programa se ejecutará y no regresa al programa original, que pudo haber terminado su ejecución. Este estilo de programación es similar al desarrollo de tareas paralelas en Unix.

```
1
2 import os
3
4 parm = 0
5 while True:
6     parm += 1
7     pid = os.fork()
8     if pid == 0:          # proceso de copia
9         os.execlp('python', 'python', 'hijo.py', str(parm))
10        assert False, 'error iniciando programa'
11    else:
12        print('Proceso hijo: ', pid)
13        if input("Salir (s)") == 's':
14            break
15
```

Listing 254. Ejemplo de uso de *os.execlp()*.

Las variantes de *os.exec* permite configurar variables de ambiente, pasar argumentos de comandos de diferentes formas, entre otras cosas.

29.2 Threads

El manejo de hilos en Python puede hacerse de con diferentes módulos. En nuestro caso, vamos a revisar 3 módulos: *thread*, *threading* y *queue*.

29.2.1 Módulo `_thread`

Este módulo es la aproximación más simple al manejo de hilos. Es básicamente una interfaz al sistema de multihilos del sistema operativo.

La función `_thread.start_new_thread` toma una función (o cualquier objeto invocable) y una tupla de argumento y e inicia un nuevo hilo para ejecutar la llamada a la función pasada como parámetro con los argumentos recibidos.

```
1  "lanza hilos hasta que se introduce 's'"
2
3  import _thread
4
5  def hijo(tid):
6      print('Hola desde el hilo', tid)
7
8  def padre():
9      i= 0
10     while True:
11         i += 1
12         _thread.start_new_thread(hijo, (i,))
13         if input("s para salir: ") == 's':
14             break
15
16 padre()
17
```

Listing 255. Ejemplo con el módulo `_thread`.

Todos los hilos se ejecutan en el mismo proceso. Un hilo puede también ejecutar una función lambda o un método enlazado de un objeto.

```
1  import _thread
2
3  def acción(i):
4      print(i ** 32)
5
6  class Potencia:
```



```
7  def __init__(self, i):
8      self.i = i
9      def acción(self):
10         print(self.i ** 32)
11
12  _thread.start_new_thread(acción, (2,))
13
14  _thread.start_new_thread((lambda: acción(2)), ())
15
16  obj = Potencia(2)
17  _thread.start_new_thread(obj.acción, ())
18
```

Listing 256. Ejemplo con el módulo `_thread` con función lambda además de un método.
El siguiente ejemplo ejecuta múltiples hilos concurrentes.

```
1  """
2  Inicia 5 copias de una función ejecutándose concurrentemente
3  Usa time.sleep para que el hilo principal no finalice muy temprano, ya que en
4  algunas plataformas esto mata a todos los otros hilos
5  La stdout es compartida, las salidas de los hilos podría verse mezclada
6  de forma arbitraria
7  """
8
9  import _thread as thread, time
10
11  def contador(miId, cont):          # función ejecutado en los hilos
12      for i in range(cont):
13          time.sleep(1)
14          print(f'[{miId}] => {i}')
15
16  for i in range(5):                # lanza 5 hilos
17      thread.start_new_thread(contador, (i, 5))    #cada hilo itera 5 veces
18
19  time.sleep(6)
20  print("Saliendo del hilo principal")
21
```

Listing 257. Ejemplo con el módulo `_thread` con 5 hilos concurrentes.

Acceso sincronizado a nombres y objetos compartidos

El módulo `_thread` incluye una forma de sincronización basada en el concepto de bloqueo (*lock*). el hilo adquiere un bloqueo, hace sus cambios, y libera el bloqueo para que otros hilos lo obtengan. Únicamente un hilo puede obtener el bloqueo en un momento en particular. Si otros hilos lo solicitan mientras se encuentra activo el bloqueo, estos son detenidos hasta que se encuentre disponible.

```
1
2 """
3 Acceso sincronizado a stdout: debido a que es compartido globalmente,
4 la salida de los hilos puede mezclarse si no se sincroniza
5 """
6
7 import _thread as thread, time
8
9 def contador(miId, cont):          # función ejecutado en los hilos
10     for i in range(cont):
11         time.sleep(1)
12         mutex.acquire()
13         print(f'[{miId}] => {i}')
14         mutex.release()
15
16 mutex = thread.allocate_lock()    # generar un objeto para bloqueo global
17 for i in range(5):                # lanza 5 hilos
18     thread.start_new_thread(contador, (i, 5))    #cada hilo itera 5 veces
19
20 time.sleep(6)
21 print("Saliendo del hilo principal")
22
```

Listing 258. Ejemplo de acceso sincronizado a stdout.

El bloqueo de hilos puede ser útil para comportamientos un poco más elaborados. El siguiente ejemplo usa una lista global de bloqueos para saber que hilos han terminado.

```
1
2 """
3 Usa mutexes para saber cuando los hilos han terminado en un hilo principal,
4 en lugar de time.sleep.
5 """
6
```

```

7  import _thread as thread
8
9  stdoutmutex = thread.allocate_lock()
10 mutexesFinalizados = [thread.allocate_lock() for i in range(10)]
11
12 def contador(miId, cont):
13     for i in range(cont):
14         stdoutmutex.acquire()
15         print(f'[{miId}] => {i}')
16         stdoutmutex.release()
17         mutexesFinalizados[miId].acquire() # signal main thread
18
19 for i in range(10):
20     thread.start_new_thread(contador, (i, 100))
21
22 for mutex in exitmutexes:
23     while not mutex.locked():
24         pass
25
26 print('Salida de hilo principal')
27

```

Listing 259. Ejemplo con una lista global para saber que un hilo ha terminado. O una versión más simple con una lista de enteros:

```

1
2  """
3  Usa una lista de datos globales (no mutexes) para saber cuando los hilos han
4  terminado en un hilo principal; lista de hilos pero no sus elementos, se asume
5  que la lista no se moverá de la memoria una vez que se ha creado inicialmente
6  """
7
8  import _thread as thread
9
10 stdoutmutex = thread.allocate_lock()
11 mutexesFinalizados = [False] * 10
12
13 def contador(miId, cont):
14     for i in range(cont):
15         stdoutmutex.acquire()
16         print(f'[{miId}] => {i}')
17         stdoutmutex.release()

```

```

18     mutexesFinalizados[miId] = True # signal main thread
19
20     for i in range(10):
21         thread.start_new_thread(contador, (i, 100))
22
23     while False in mutexesFinalizados:
24         pass
25
26     print('Salida de hilo principal')
27

```

Listing 260. Ejemplo con una lista de enteros para saber que un hilo ha terminado.

29.2.2 Módulo threading

El módulo `threading` es una interface de alto nivel basada en clases y objetos. Internamente usa el módulo `_thread` para implementar objetos que representen hilos y herramientas básicas de sincronización.

```

1
2     """
3     Instancias de clase thread con estado y run() para la acción del hilo;
4     usa módulo multihilo de alto nivel similar a Java con método join (sin mutexes ni variab
5     compartidas globales) para saber cuando los hilos han terminado en el
6     hilo principal.
7
8     """
9     import threading
10
11     class MiHilo(threading.Thread):
12         def __init__(self, miId, cont, mutex):
13             self.miId = miId
14             self.cont = cont
15             self.mutex = mutex
16             threading.Thread.__init__(self)
17
18         def run(self):
19             for i in range(self.cont):
20                 with self.mutex:
21                     print(f'[{self.miId}] => {i}')
22
23     stdoutmutex = threading.Lock()

```

```
24 hilos = []
25 for i in range(10):
26     hilo = MiHilo(i, 100, stdoutmutex)
27     hilo.start()
28     hilos.append(hilo)
29
30 for hilo in hilos:
31     hilo.join()
32
33 print('saliendo del hilo principal.')
34
```

Listing 261. Ejemplo de uso de módulo *threading*.

En el ejemplo anterior vemos como se usa el método *join()* para esperar a que el hilo salga; este método puede ser usado para prevenir que se salga del hilo principal antes de los hilos hijos, en lugar de usar *time.sleep()*. También se puede usar *threading.lock()* para sincronizar el acceso al flujo de manera similar a *_thread.allocate_lock*

En el siguiente ejemplo se muestra como se puede usar la clase *Thread* para iniciar una función simple, o de hecho cualquier tipo de objeto invocable, sin el uso de subclases.

```
1
2 import threading, _thread
3
4 def accion(i):
5     print(i ** 32)
6
7 # subclase con estado
8 class MiHilo(threading.Thread):
9     def __init__(self, i):
10         self.i = i
11         threading.Thread.__init__(self)
12
13     def run(self):
14         print(self.i ** 32)
15
16 MiHilo(2).start()
17
18 # pasar acción en
19 hilo = threading.Thread(target=(lambda: accion(2)))
```

```
20 hilo.start()
21
22 #lo mismo pero sin envoltura lambda para el estado
23 threading.Thread(target=accion, args=(2,)).start() # invocable mas sus args
24
25 #simple módulo de hilo
26 _thread.start_new_thread(accion, (2,))
27
```

Listing 262. Ejemplo de uso de módulo *threading* usando la clase *Thread* sin herencia.

Generalmente, hilos basados en clases pueden ser mejores si los hilos requieren un estado por hilo o pueden aprovechar los beneficios del paradigma OO en general. Las clases de hilos no necesariamente tienen que ser una subclase de *Thread*. Es posible inclusive combinar técnicas de clases anidadas y métodos enlazados.

```
1
2 import threading, _thread
3 #una clase sin heredar de hilo con su estado
4
5 class Potencia:
6     def __init__(self, i):
7         self.i = i
8
9     def accion(self):
10        print(self.i ** 32)
11
12 obj = Potencia(2)
13 threading.Thread(target=obj.accion).start()
14
15 # alcance anidado para retener el estado
16 def accion(i):
17     def potencia():
18         print(i ** 32)
19     return potencia
20
21 threading.Thread(target=accion(2)).start()
22
23 # ambas con módulo básico de hilo
24 _thread.start_new_thread(obj.accion, ())
25 _thread.start_new_thread(accion(2), ())
26
```

Listing 263. Ejemplo de uso de módulo *threading* con referencia anidada.

29.2.3 Módulo Queue

Programas con hilos pueden ser frecuentemente estructurados como un conjunto de hilos productores y consumidores, que se comunican colocando datos y tomándolos de una cola compartida. Mientras la cola sincronice el acceso a sí misma, esto automáticamente sincronizará las interacciones entre los hilos.

Python cuenta con el módulo *queue* que proporciona una estructura estándar de cola para objetos en el que los elementos son añadidos por un lado y extraídos por el otro. La diferencia es que la cola de objetos es automáticamente controlada con operaciones de bloqueo de hilo y liberación, de forma que únicamente un hilo pueda modificar la cola en un momento determinado.

El siguiente ejemplo muestra 2 hilos consumidores que esperan por datos que aparecen en la cola compartida y 4 hilos productores que colocan datos en la cola periódicamente después de dormir por un intervalo de tiempo. De forma que este ejemplo está ejecutando 7 hilos concurrentes (incluyendo el hilo principal) donde 6 hilos acceden a la cola compartida de forma concurrente.

```
1
2 # hilos productor y consumidor comunicándose con una cola compartida
3
4 numconsumidores = 2          # cantidad de consumidores para iniciar
5 numproductores = 4           # cantidad de productores para iniciar
6 nummensajes = 4              # mensajes para poner por productor
7
8 import _thread as thread, queue, time
9
10 print_seguro = thread.allocate_lock() # prints en el else se podrían superponer
11 cola_datos = queue.Queue()           # compartido global, tamaño infinito
12
13 def productor(idnum, cola_datos):
14     for num_mensaje in range(nummensajes):
15         time.sleep(idnum)
16         cola_datos.put('[productor id=%d, contador=%d]' % (idnum, num_mensaje))
17
18 def consumidor(idnum, cola_datos):
19     while True:
20         time.sleep(0.1)
21         try:
22             dato = cola_datos.get(block=False)
23             except queue.Empty:
```

```
24     pass
25     else:
26         with print_seguro:
27             print('consumidor', idnum, 'obtuvo =>', dato)
28
29 if __name__ == '__main__':
30     for i in range(numconsumidores):
31         thread.start_new_thread(consumidor, (i, cola_datos))
32     for i in range(numproductores):
33         thread.start_new_thread(producer, (i, cola_datos))
34     time.sleep(((numproductores-1) * nummensajes) + 1)
35     print('Salida de hilo principal.')
36
```

Listing 264. Ejemplo de uso de módulo *queue* productor/consumidor.

Capítulo 30

Java: Clases anidadas y anónimas

30.1 Clases anidadas

Las clases anidadas son clases definidas dentro del alcance de otras clases. Existen dos tipos de clases anidadas¹:

- Clase anidadas estáticas
- Clases interiores, las cuales son clases anidadas no estáticas

Sintaxis

```
class claseExterna {  
    ...  
    static class ClaseAnidadaEstática {  
        ...  
    }  
    class claseInterior {  
        ...  
    }  
}
```

Una clase anidada es un miembro de la clase que la define, la clase externa. Clases interiores (clases anidadas no estáticas) tienen acceso a otros miembros de la clase externa, inclusive si estos miembros son privados. Clases anidadas estáticas

¹Fuente: [Nested class](#)

no tienen acceso a los otros miembros de la clase externa. Las clases anidadas, al ser miembros de una clase pueden ser definidas como públicas, protegidas o privadas (privadas a nivel de paquete).

Una clase anidada puede ser

- Es una forma de agrupar clases que son usadas en un sólo lugar
- Incrementa la encapsulación
- Puede llevar a un código más fácil de leer y/o mantener.

Ejemplo: Clases internas.

```
1 public class EstructuraDeDatos {
2
3     // Crea un arreglo
4     private final static int TAMAÑO = 15;
5     private int[] arregloDeEnteros = new int[TAMAÑO];
6
7     public EstructuraDeDatos() {
8         for (int i = 0; i < TAMAÑO; i++) {
9             arregloDeEnteros[i] = i;
10        }
11    }
12
13    public void despliegaPares() {
14
15        // despliega valores de índices pares del arreglo
16        IteradorDeEstructuraDeDatos iterador = this.new IteradorPares();
17        while (iterador.hasNext()) {
18            System.out.print(iterador.next() + " ");
19        }
20        System.out.println();
21    }
22
23    //interfaz anidada extiende la interfaz Iterator<Integer>
24    interface IteradorDeEstructuraDeDatos extends java.util.Iterator<Integer> { }
25
26    // Clase anidada implementa interfaz IteradorDeEstructuraDeDatos
27    private class IteradorPares implements IteradorDeEstructuraDeDatos {
28
29        // Inicializa el iterador para el inicio del arreglo
30        private int nextIndice = 0;
```

```
32         @Override
33         public boolean hasNext() {
34             //Verifica si el elemento actual es el último en el arreglo
35             return (nextIndice <= TAMAÑO - 1);
36         }
37
38         @Override
39         public Integer next() {
40             // Registra un valor de un índice par del arreglo
41             Integer reValor = Integer.valueOf(arregloDeEnteros[nextIndice]);
42
43             // obtiene el siguiente elemento par
44             nextIndice += 2;
45             return reValor;
46         }
47
48         @Override
49         public void remove() {
50             // implementación es opcional
51             throw new UnsupportedOperationException();
52         }
53     }
54 }
55
56 public static void main(String s[]) {
57     EstructuraDeDatos edd = new EstructuraDeDatos();
58     edd.despliegaPares();
59 }
60 }
```

Listing 265. Ejemplo de clases internas.

Variables con el mismo nombre. [Ejemplo:](#)

```
1 public class PruebaOcultarVariables {
2
3     public int x = 0;
4
5     class PrimerNivel {
6
7         public int x = 1;
8     }
```

```

9      void métodoEnPrimerNivel(int x) {
10          System.out.println("x = " + x);
11          System.out.println("this.x = " + this.x);
12          System.out.println("PruebaOcultarVariables.this.x = "
13              + PruebaOcultarVariables.this.x); //Despliega x de la clase externa (x=0)
14      }
15  }
16
17  public static void main(String... args) {
18      PruebaOcultarVariables pov = new PruebaOcultarVariables();
19      PruebaOcultarVariables.PrimerNivel pn = pov.new PrimerNivel();
20      pn.métodoEnPrimerNivel(25);
21  }
22  }

```

Listing 266. Ejemplo variables con el mismo nombre en clases internas .

30.2 Clases anónimas

Una clase anónima es una clase anidada que no tiene un nombre. Combina en un solo paso la definición de la clase anidada y la instanciación de la misma.

Una clase anónima se ocupa cuando una clase solo se requiere una vez².

Las clases anónimas son compiladas con el nombre de las clase que las contienen seguida de \$ y un número consecutivo. Por ejemplo: *Clasecontenedora\$1.class*

Un código de clases anónimas. [Ejemplo:](#)

```

1  public class ClasesAnónimasHolaMundo {
2
3      interface HolaMundo {
4          public void saludar();
5          public void saludarAlguien(String alguien);
6      }
7
8      public void decirHola() {
9
10         class SaludarEnInglés implements HolaMundo {
11             String nombre = "world";
12             public void saludar() {
13                 saludarAlguien("world");

```

²[Anonymouse classes](#)

```
14         }
15         public void saludarAlguien(String alguien) {
16             nombre = alguien;
17             System.out.println("Hello " + nombre);
18         }
19     }
20
21     HolaMundo saludarEnInglés = new SaludarEnInglés();
22
23     HolaMundo saludarEnFrances = new HolaMundo() {
24         String nombre = "tout le monde";
25         public void saludar() {
26             saludarAlguien("tout le monde");
27         }
28         public void saludarAlguien(String alguien) {
29             nombre = alguien;
30             System.out.println("Salut " + nombre);
31         }
32     };
33
34     HolaMundo saludarEnEspañol = new HolaMundo() {
35         String nombre = "mundo";
36         public void saludar() {
37             saludarAlguien("mundo");
38         }
39         public void saludarAlguien(String alguien) {
40             nombre = alguien;
41             System.out.println("Hola, " + nombre);
42         }
43     };
44
45     saludarEnInglés.saludar();
46     saludarEnFrances.saludarAlguien("Juliette");
47     saludarEnEspañol.saludar();
48 }
49
50 public static void main(String... args) {
51     ClasesAnónimasHolaMundo miApl =
52     new ClasesAnónimasHolaMundo();
53     miApl.decirHola();
54 }
```

55

```
}
```

Listing 267. Ejemplo de clases anónimas.

Capítulo 31

Java: Expresiones lambda λ

31.1 Introducción

Java fue creado como un lenguaje orientado a objetos en los 90's, pero la tendencia de los lenguajes actuales es ser multiparadigma, tomando conceptos y adaptándolos a los lenguajes actuales. Un paradigma que antes estaba confinado más al mundo académico es el de programación funcional. Éste ha tomado mayor relevancia porque funciona bien con programación concurrente o manejada por eventos. Ejemplos de lenguajes puramente funcionales son Haskell y Erlang.



Java 8 añade constructores de programación funcional a sus bases orientadas a objetos. Algunos puntos básicos son¹:

- Una expresión lambda es un bloque de código con parámetros.
- Expresiones lambda pueden ser convertidas en interfaces funcionales
- Estas expresiones pueden acceder efectivamente variables finales dentro del alcance que encierran.

¹[Lambda expressions in Java](#)

- Se puede tener ahora métodos default y estáticos en las interfaces que ofrecen implementación concreta. Estos métodos de múltiples interfaces pueden generar conflictos que debe atender el programador.

Una expresión lambda es un bloque de código que puede ser pasado y ejecutarse eventualmente, una vez o múltiples veces. Básicamente permite la escritura de un método en el lugar que necesitas usarlo. Práctico especialmente si el método solo se va a usar una vez y éste es corto.

Pasar un bloque de código no era fácil en Java. Al ser orientado a objetos, se tenía que construir un objeto que perteneciera a una clase que tuviera el método con el código necesario.

El nombre lambda viene del lógico y matemático **Alonzo Church**, el cual quería formalizar lo que significa para una función matemática ser efectivamente computable. El usó la letra Griega minúscula lambda λ para marcar los parámetros. Propuso entonces un sistema formal conocido como cálculo lambda ($\lambda - calculus$).

$\lambda - calculus$ es un simple modelo conceptual de cómputo universal. De hecho, Turing demostró en 1937 que las máquinas de Turing tienen una expresividad equivalente a $\lambda - calculus$ [?].

31.2 Sintaxis de expresiones lambda

La sintaxis general de una expresión lambda en Java es:

Sintaxis
(<argumentos>) -> <cuerpo>

Por ejemplo, las siguientes son expresiones lambda válidas:

```

1 (int a, int b) -> {return a+b;}
2
3 ( ) -> System.out.println("Lambda")
4
5 (String s) -> System.out.println(s)
6
7 ( ) -> 123

```

Algunas características relevantes²:

- Una expresión puede tener de cero a n parámetros.

²[Lambda expressions java tutorial](#)

- El tipo del parámetro puede indicarse explícitamente o puede ser inferido del contexto.
- Cuando no hay parámetros se debe indicar con paréntesis vacíos (). Si se tienen un sólo parámetro y su tipo es inferido, los paréntesis son opcionales.
- El cuerpo de la expresión lambda puede ir vacío.
- Si se tiene una sola instrucción en el cuerpo, se pueden omitir las llaves y el tipo de retorno de la función es el mismo que el de la instrucción.

Las expresiones lambda pueden ser usadas en lugar de las clases anónimas para implementar el método de una interfaz funcional. Una **interfaz funcional** es una interfaz que tiene sólo un método abstracto declarado. Cada expresión lambda puede ser implícitamente asignada a una interfaz funcional.

Por ejemplo, inclusive cuando no especificamos la interfaz funcional, el compilador automáticamente resuelve que la siguiente expresión lambda puede ser enmascarada a la interfaz *Runnable* en la firma del constructor *Thread(Runnable r)*:

```
1 new Thread(  
2     ( ) -> System.out.println("ejemplo de expresión lambda")  
3 ).start();
```

Dos diferencias importantes entre expresiones lambda y clases anónimas:

- Para una clase anónima el uso de *this* se refiere a la clase anónima, mientras que el uso de *this* en una expresión lambda se refiere a la clase que contiene la expresión lambda.
- Para el compilador, una clase anónima es tratada como una clase y generará su código respectivo, mientras que una expresión lambda es compilada y convertida en un método privado de la clase que contiene la expresión.

Capítulo 32

Java: Interfaz gráfica con JavaFX

32.1 Introducción

JavaFX es el framework más reciente para el desarrollo de interfaces gráficas en Java. Como ya se mencionó, la AWT (*Abstract Window Toolkit*) es útil para desarrollar aplicaciones con interfaces gráficas simples, pero es dependiente de la plataforma. AWT fue reemplazada por Swing - fue incluida inicialmente en Java 1.1-, incluyendo componentes más robustos que son puestos en lienzos (canvas) y estos componentes son menos dependientes de la plataforma de ejecución, usando menos recursos nativos¹. Swing incluyó componentes ligeros construidos enteramente en Java y *look & feel* independiente de la plataforma, separando la vista de la lógica del componente.

JavaFX es un conjunto nuevo de componentes para interfaces gráficas que permite desarrollar RIA (*Rich Internet Applications*). Una aplicación RIA es una aplicación Web que ofrece las mismas características que una aplicación de escritorio. Agrega soporte *multi-touch* (para tabletas y *smartphones*), animación 2D y 3D, reproducción de audio, video, etc.[?]

Una aplicación JavaFX puede ejecutarse²:

- Como aplicación de escritorio desde un archivo JAR
- Desde la línea de comandos usando el lanzador JavaBeans
- Dando click en un navegador y bajando la aplicación
- En una página web al abrirse.

La arquitectura de JavaFX puede apreciarse en la figura ??³.

¹Información sobre JavaFX y Swing: [Java SE client technologies](#)

²[Basic deployment](#)

³<https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm>

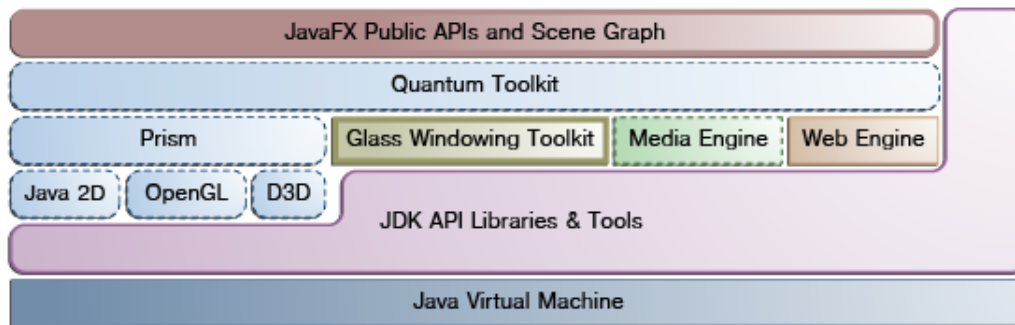


Fig. 32.1. Arquitectura de JavaFX

El framework de JavaFX está contenido con el prefijo *javafx*, contando con más de 30 paquetes es la API de Java. La estructura general de una interfaz de usuario JavaFX está basada en:

- *Stage* (Escenario)
- *Scene* (Escena)
- *Node*
- *Layout* (Esquema, disposición)

JavaFX utiliza una **metáfora de escenario**. En una obra teatral un escenario contiene a una escena. De esta forma un escenario define un espacio y una escena define que va en ese espacio. Podríamos decir que un escenario es un contenedor de escenas y una escena es un contenedor de los elementos relacionados con dicha escena.

Una aplicación JavaFX tiene al menos un escenario y una escena mediante las clases *Stage* y *Scene*. *Stage* es un contenedor de alto nivel llamado **escenario primario**. Es posible tener más de un escenario pero solo un escenario primario es requerido.

Los elementos de una escena son nodos. Puede tratarse de un elemento único (e.g., un botón) hasta grupos de nodos. Adicionalmente, un nodo puede tener un nodo hijo. Por lo que podemos tener: nodos padre, nodos hijo, hojas (o nodos terminales) y el nodo raíz (el nodo en el nivel más alto del árbol). La clase base para todos los nodos es *Node*, algunas de sus principales subclases son *Parent*, *Group*, *Region* y *Control*.

Los *Layouts* son esquemas que se utilizan para organizar el contenido de los elementos en una escena. Se tienen diferentes clases de “páneles” que heredan de la clase *Node*.

El siguiente [ejemplo](#) es generado por default en NetBeans al crear un proyecto JavaFX:

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {
7
8      @Override // Sobreescribe el método start en la clase Application
9      public void start(Stage primaryStage) {
10         // Crea una escena y coloca un botón en la escena
11         Button btOK = new Button("OK");
12         Scene scene = new Scene(btOK, 200, 250);
13         primaryStage.setTitle("MyJavaFX");
14         primaryStage.setScene(scene); // Coloca escena en escenario
15         primaryStage.show(); // Despliega escenario
16     }
17
18     /**
19      * El método main solo es necesario para IDEs con soporte limitado
20      * de JavaFX. No necesario para ejecución en línea de comandos.
21      */
22     public static void main(String[] args) {
23         Application.launch(args);
24     }
25 }
```

Listing 268. Ejemplo JavaFX generado por NetBeans.
Otro [ejemplo](#)[?]

```
1  package javafxapplication1;
2
3  import javafx.application.Application;
4  import javafx.event.ActionEvent;
5  import javafx.event.EventHandler;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Button;
8  import javafx.scene.layout.StackPane;
9  import javafx.stage.Stage;
10
11
12  public class JavaFXApplication1 extends Application {
```

```
13
14     @Override
15     public void start(Stage escenarioPrimario) {
16         Button btn = new Button();
17         btn.setText("Di 'Hola Mundo'");
18         btn.setOnAction(new EventHandler<ActionEvent>() {
19
20             @Override
21             public void handle(ActionEvent event) {
22                 System.out.println("¡Hola, Mundo!");
23             }
24         });
25
26         StackPane root = new StackPane();
27         root.getChildren().add(btn);
28
29         Scene escena = new Scene(root, 300, 250);
30
31         escenarioPrimario.setTitle("¡Hola Mundo!");
32         escenarioPrimario.setScene(escena);
33         escenarioPrimario.show();
34     }
35
36     /**
37      * @param args the command line arguments
38      */
39     public static void main(String[] args) {
40         launch(args);
41     }
42
43 }
```

Listing 269. Ejemplo JavaFX "¡Hola, Mundo!".

El método *launch()* es un método estático definido en la clase *Application* y que se ocupa para lanzar la aplicación *stand-alone* de JavaFX. El método *main(...)* no es necesario si se ejecuta de la terminal, pero puede ser necesario para ejecutar el programa en un IDE que no tenga soporte completo para JavaFX. Al ejecutarse la aplicación JavaFX sin el método *main*, la máquina virtual de Java ejecuta el método *run()* de la aplicación.

En este ejemplo, la clase sobrescribe el método *start()* originalmente definido en:

javafx.application.Application

Al ejecutarse el programa, la máquina virtual genera una instancia de la clase usando el constructor sin parámetros e invoca a su método *start()*.

El método *start()* usualmente se encarga de colocar los controles de la interfaz de usuario en una escena (*scene*) y la despliega en un escenario (*stage*).

El ejemplo crea un objeto *Button* y lo coloca en un objeto *Scene*. El objeto *Scene* puede ser creado con el constructor *Scene(node, width, height)*. Se especifica el ancho y alto de la escena y coloca el nodo en la escena.

Un objeto *Stage* es una ventana. Un objeto *Stage* llamado *primaryStage* es creado automáticamente por la JVM al lanzarse la aplicación.

Recordemos que JavaFX usa una **analogía de un teatro** con clases de escenas y escenarios. Puede verse a un escenario como una plataforma que soporta escenas y nodos como actores que intervienen en las escenas.

Un [ejemplo\[?\]](#) con múltiples escenarios, se omite el método *main* ya que, como se explicó no debe ser necesario:

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MúltipleEscenario extends Application {
7      @Override
8      public void start(Stage escenarioPrimario) {
9          // Crea una escena y coloca un botón en la escena
10         Scene escena = new Scene(new Button("OK"), 200, 250);
11         escenarioPrimario.setTitle("MúltipleEscenarioApp"); // asigna título al escenario
12         escenarioPrimario.setScene(escena); // Coloca la escena en el escenario
13         escenarioPrimario.show(); // Despliega el escenario
14
15         Stage escenario = new Stage(); // Crea un nuevo escenario
16         escenario.setTitle("Segundo escenario");
17         // Coloca una escena con un botón en el escenario
18         escenario.setScene(new Scene(new Button("Nuevo escenario"), 100, 100));
19         escenario.show();
20     }
21 }
```

Listing 270. Ejemplo de JavaFX con múltiples escenarios.

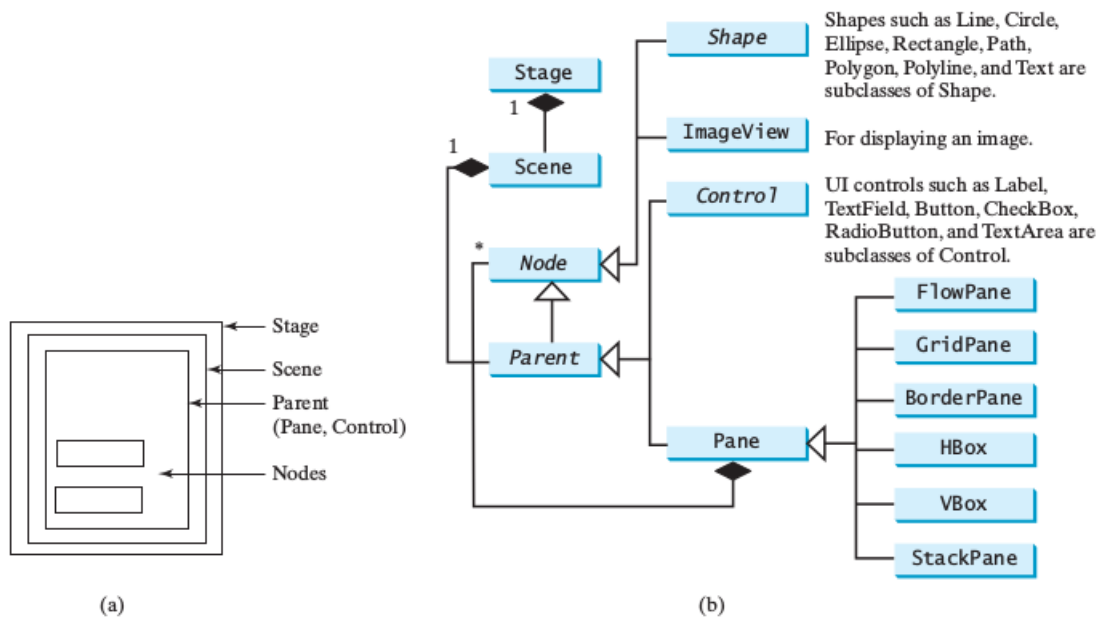


Fig. 32.2. (a) Paneles son usados para mantener nodos. (b) Nodos pueden ser figuras, vistas de imágenes, controles IU, y paneles.

32.2 Elementos de interfaz de usuario

Los elementos de interfaz con el usuario se pueden posicionar estableciendo las posiciones y el tamaño de los elementos en la ventana, pero esto no es la mejor solución. Un modelo más flexible es usar clases contenedoras de tipo panel (*pane*) para colocar los nodos. Un nodo es un componente visual que puede ser un control de interfaz de usuario, una figura o un panel. Se colocan los nodos en un panel y se coloca el panel en una escena. La escena puede ser mostrada en un escenario, como ya se vió en los ejemplos anteriores. Ver figura ?? [?].

Un ejemplo [?] con panel:

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5  import javafx.scene.layout.StackPane;
6
7  public class BotónEnPanel extends Application {
8      @Override
9      public void start(Stage primaryStage) {
10         // Crea una escena y coloca botón en la escena
  
```



```
11     StackPane panel = new StackPane();
12     panel.getChildren().add(new Button("OK"));
13     Scene escena = new Scene(panel, 200, 50);
14     primaryStage.setTitle("Botón en un panel");
15     primaryStage.setScene(escena);
16     primaryStage.show();
17 }
18 }
```

Listing 271. Ejemplo de JavaFX con panel.

Aunque aquí solo hay un objeto, un objeto *StackPane* coloca los nodos en el centro del panel y los apila, respetando el tamaño preferido del nodo.

32.2.1 Diseño de paneles

De forma similar a las otras bibliotecas de IU, JavaFX ofrece un conjunto de tipos de paneles para posicionar automáticamente los nodos en un tamaño y posición necesarios. A continuación se describen de manera general los principales paneles manejados.

- *Pane*. Es la clase base de diseño de paneles. Contiene el método *getChildren()* que regresa la lista de nodos del panel.
- *StackPane*. Coloca los nodos encima unos de otros en el centro del panel.
- *FlowPane*. Coloca los nodos renglón por renglón de manera horizontal o columna por columna, verticalmente.
- *GridPane*. Coloca los nodos en celdas en una tabla de dos dimensiones.
- *BorderPane*. Coloca los nodos en las zonas de arriba, derecha, abajo, izquierda y centro.
- *HBox*. Coloca los nodos en un renglón simple.
- *VBox*. Coloca los nodos en una columna sencilla.

Los nodos se agregan a la lista del panel con el método *add(< nodo >)* de forma individual o con el método *addAll(< nodo1 >, < nodo2 >, ..., < nodo_n >)* para añadir un conjunto de nodos al panel.

Usando *FlowPane*. [Ejemplo:](#)

```
1 import javafx.application.Application;
2 import javafx.geometry.Insets;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Label;
5 import javafx.scene.control.TextField;
6 import javafx.scene.layout.FlowPane;
7 import javafx.stage.Stage;
8
9 public class MuestraFlowPane extends Application {
10     @Override
11     public void start(Stage escenarioPrimario) {
12         // Crea un panel y asigna sus propiedades
13         FlowPane panel = new FlowPane();
14         // asigna las propiedades de relleno de espacio
15         // con un objeto de Insets
16         // Construye una nueva instancia Insets
17         // con cuatro diferentes offsets .
18         // Parámetros: top - the top offset;
19         // right - the right offset;
20         // bottom - the bottom offset; left - the left offset.
21         // Fuente:
22         // https://docs.oracle.com/javase/8/javafx/api/javafx/geometry/Insets.h
23         panel.setPadding(new Insets(11, 12, 13, 14));
24         panel.setHgap(5);
25         panel.setVgap(5);
26         // Coloca los nodos en el panel
27         panel.getChildren().addAll(new Label("Nombre:"),
28             new TextField());
29         TextField tfIniciales = new TextField();
30         tfIniciales.setPrefColumnCount(2);
31         panel.getChildren().addAll(new Label("Apellidos:"),
32             new TextField(), new Label("Iniciales:"), tfIniciales);
33         // Crea una escena y la coloca en el escenario
34         Scene escena = new Scene(panel, 200, 250);
35         escenarioPrimario.setTitle("MuestraFlowPane");
36         escenarioPrimario.setScene(escena);
37         escenarioPrimario.show();
38     }
39 }
```

Listing 272. Ejemplo JavaFX de panel usando *FlowPane*.

Usando `BorderPane`. [Ejemplo:](#)

```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.BorderPane;
6  import javafx.scene.layout.StackPane;
7  import javafx.stage.Stage;
8
9  public class MuestraBorderPane extends Application {
10     @Override
11     public void start(Stage escenarioPrimario) {
12         // Crea un BorderPane
13         BorderPane panel = new BorderPane();
14         // Coloca nodos en el panel
15         panel.setTop(new PanelAdaptado("Arriba"));
16         panel.setRight(new PanelAdaptado("Derecha"));
17         panel.setBottom(new PanelAdaptado("Abajo"));
18         panel.setLeft(new PanelAdaptado("Izquierda"));
19         panel.setCenter(new PanelAdaptado("Centro"));
20
21         // Crea una escena y la coloca en el escenario
22         Scene escena = new Scene(panel);
23         escenarioPrimario.setTitle("MuestraBorderPane");
24         escenarioPrimario.setScene(escena);
25         escenarioPrimario.show();
26     }
27 }
28
29 // Define un panel adaptado para mantener una etiqueta en el centro del panel
30 class PanelAdaptado extends StackPane {
31     public PanelAdaptado(String title) {
32         getChildren().add(new Label(title));
33         setStyle("-fx-border-color: red");
34         setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
35     }
36 }
```

Listing 273. Ejemplo JavaFX usando `BorderPane`.

Resumiendo:

- JavaFX es un framework para interfaces de usuario de aplicaciones tanto

stand-alone como web

- La idea es reemplazar *AWT* y *Swing*.
- Una clase principal de JavaFX debe extender la clase *javafx.application.Application* e implementar el método *start()*.
- El escenario primario es creado automáticamente por la JVM y pasado al método *start()*.
- Un escenario es una ventana para el desplegado de una escena. Se pueden añadir nodos a una escena. Paneles, controles y figuras son nodos. Paneles pueden ser usados como contenedores de nodos.
- La clase *Node* define muchas propiedades que son comunes a todos los nodos. Estas propiedades pueden ser aplicadas a paneles, controles y figuras.

Capítulo 33

Java: Manejo de eventos con JavaFX

33.1 Introducción

Al igual que con *AWT* o *Swing*, no se trata nada más del diseño de la interfaz de usuario. También se tiene que considerar como se incorpora el comportamiento que el usuario tiene sobre la interfaz, esto se conoce como manejo de eventos.

Por ejemplo, al dar *click* a una aplicación se necesita incorporar código que procese esa acción. El botón entonces es el **objeto fuente del evento** donde la acción se origina, el evento por si solo es un objeto. Es necesario crear un objeto capaz de manejar el evento de acción sobre un botón. Este objeto es llamado un manejador de eventos (*event handler*). Ver figura ??

Para poder ser manejador de un evento de acción se requiere:

- El objeto debe ser una instancia de la interfaz *EventHandler* $\langle T \text{ extends } Event \rangle$. Esta interfaz define el comportamiento común para todos los manejadores.

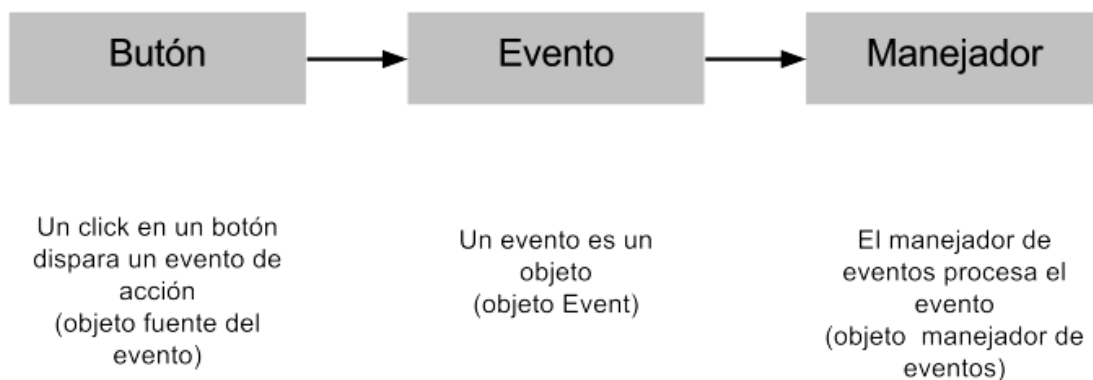


Fig. 33.1. Proceso de manejo de evento

Por su parte, *<T extends Event>* especifica que *T* es un tipo genérico que es un subtipo de *Event*.

- El objeto manejador de *EventHandler* debe estar registrado con el objeto fuente del evento con el método *fuelle.setOnAction(manejador)*.

La interfaz *EventHandler<ActionEvent>* contiene el método *handle(ActionEvent)* para procesar el evento de acción. Nuestra clase manejadora debe sobrescribir este método para responder al evento.

El siguiente [ejemplo](#) procesa el evento de acción para 2 botones y despliega el mensaje correspondiente al ser presionados:

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.layout.HBox;
6  import javafx.stage.Stage;
7  import javafx.event.ActionEvent;
8  import javafx.event.EventHandler;
9
10
11 public class ManejoDeEventos extends Application {
12     @Override
13     public void start(Stage escenarioPrimario) {
14
15         // Crea un panel y establece sus propiedades
16         HBox panel = new HBox(10);
17         panel.setAlignment(Pos.CENTER);
18         Button btHola = new Button("Hola");
19         Button btAdiós = new Button("Adiós");
20         ClaseManejadoraHola manejador1 = new ClaseManejadoraHola();
21         btHola.setOnAction(manejador1);
22         ClaseManejadoraAdiós manejador2 = new ClaseManejadoraAdiós();
23         btAdiós.setOnAction(manejador2);
24         panel.getChildren().addAll(btHola, btAdiós);
25
26         // Crea una escena y la coloca en el escenario
27         Scene escena = new Scene(panel);
28         escenarioPrimario.setTitle("ManejoDeEventos");
29         escenarioPrimario.setScene(escena);
30         escenarioPrimario.show();
31     }
```

```
32 }
33
34 class ClaseManejadoraHola implements EventHandler<ActionEvent> {
35     @Override
36     public void handle(ActionEvent e) {
37         System.out.println("Hola");
38     }
39 }
40
41 class ClaseManejadoraAdiós implements EventHandler<ActionEvent> {
42     @Override
43     public void handle(ActionEvent e) {
44         System.out.println("Adiós");
45     }
46 }
```

Listing 274. Ejemplo JavaFX manejo de evento acción para 2 botones .

Como se aprecia, se declararon dos clases manejadoras, cada una implementa *EventHandler<ActionEvent>* para procesar el evento de acción *ActionEvent*. El objeto *manejador1* es una instancia de *ClaseManejadoraHola*, y es registrado en el botón *btHola*. Cuando el botón es presionado, el método *handle(ActionEvent)* en *ClaseManejadoraHola* es invocado para procesar el evento. El mismo proceso se sigue para el otro botón.

33.2 Más sobre eventos

Un evento es un objeto creado desde la fuente de un evento. Disparar un evento significa crear un evento y delegarlo al manejador para que maneje el evento.

Al ejecutar una interfaz gráfica en Java, el programa interactúa con el usuario y los eventos conducen la ejecución. Esto se conoce como programación manejada por eventos (*event-driven*). Un evento puede ser visto como una señal para el programa de que algo ha pasado. Los eventos son disparados por acciones externas del usuario, como movimientos del *mouse*, *clicks*, teclas presionadas. Un programa puede reaccionar o ignorar los eventos.

El componente que crea un evento y lo dispara es el objeto fuente u origen del evento. Un evento es una instancia de una clase de evento. La clase raíz de las clases evento en Java es *java.util.EventObject*. Pero la clase raíz de las clases de evento en JavaFX es *javafx.event.Event*, por lo que un evento en JavaFX es un objeto de esta clase o una de sus subclases. Algunas de las cuales se muestran a continuación:

- *EventObject*

- Event
 - ActionEvent
 - InputEvent
 - ◊ MouseEvent
 - ◊ KeyEvent
 - WindowEvent

Un objeto de evento contiene propiedades propias del tipo de evento que maneja. Un objeto de evento puede identificar su origen mediante el método `getSource()`. Como se puede ver, las subclases de evento tratan con específicos tipos de eventos.

33.3 Registro de manejadores y manejo de eventos

Un manejador es un objeto que debe ser registrado en el objeto fuente del evento, y debe ser una instancia de una interfaz apropiada de manejo de eventos.

Java usa un modelo basado en delegación para el manejo de eventos: un objeto fuente dispara un evento, y un objeto interesado en el evento lo maneja. Este último evento es llamado manejador de eventos (*event handler*) o escuchador de eventos (*event listener*). Para que un objeto sea un manejador de un evento para un objeto fuente se tiene que cumplir:

- El objeto manejador debe ser una instancia de la correspondiente interfaz manejadora de evento, para asegurar que el manejador tiene el método correcto para procesar el evento.
 - JavaFX tiene definida una interfaz manejadora unificada *EventHandler* `<T extends Event>` para un evento *T*.
 - La interfaz manejadora contiene el método *handle(T e)* para procesar el evento.
 - Por ejemplo, la interfaz manejadora para *ActionEvent* es *EventHandler* `<ActionEvent>`; cada manejador para *ActionEvent* debe implementar el método *handle(ActionEvent e)* para procesar un *ActionEvent*.
- El objeto manejador debe ser registrado por el objeto fuente. Los métodos a registrar dependen del tipo de evento.
 - Para *ActionEvent* el método es *setOnAction()*.
 - Para el evento de presionar una tecla, el método es *setOnKeyPressed()*.
 - Para el evento del mouse presionado, el método es *setOnMousePressed()*.

Vamos a ver ahora un [ejemplo](#) donde dos botones controlan el tamaño de un círculo. Vemos el código primero sin manejo de eventos:

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.layout.StackPane;
6  import javafx.scene.layout.HBox;
7  import javafx.scene.layout.BorderPane;
8  import javafx.scene.paint.Color;
9  import javafx.scene.shape.Circle;
10 import javafx.stage.Stage;
11
12 public class ControlDeCírculoAúnSinManejoDeEventos extends Application {
13     @Override
14     public void start(Stage escenarioPrimario) {
15         StackPane panel = new StackPane();
16         Circle círculo = new Circle(50);
17         círculo.setStroke(Color.BLACK);
18         círculo.setFill(Color.WHITE);
19         panel.getChildren().add(círculo);
20         HBox hBox = new HBox();
21         hBox.setSpacing(10);
22         hBox.setAlignment(Pos.CENTER);
23         Button btAumentar = new Button("Aumentar");
24         Button btReducir = new Button("Reducir");
25         hBox.getChildren().add(btAumentar);
26         hBox.getChildren().add(btReducir);
27
28         BorderPane borderPanel = new BorderPane();
29         borderPanel.setCenter(panel);
30         borderPanel.setBottom(hBox);
31         BorderPane.setAlignment(hBox, Pos.CENTER);
32
33         Scene escena = new Scene(borderPanel, 200, 150);
34         escenarioPrimario.setTitle("ControlDeCírculo");
35         escenarioPrimario.setScene(escena);
36         escenarioPrimario.show();
37     }
38 }
```

Listing 275. Ejemplo JavaFX dos botones sin manejo de eventos.

Para lograr las acciones deseadas vamos a introducir los siguientes cambios:

- Definir una clase *PanelCírculo* para desplegar el círculo en un panel, proporcionando métodos para alagar y reducir modificando el radio del círculo.
- Crear un objeto de *PanelCírculo* y declarar un atributo que haga referencia a este objeto en la clase *ControlDeCírculo*. Los métodos en esta clase ahora accesan al objeto de *PanelCírculo* a través de este atributo.
- Definir una clase manejadora *AgrandarManejador* que implementa *EventHandler <ActionEvent>*. Para hacer accesible la variable de referencia *PanelCírculo* desde el método *handle()*, se define *AgrandarManejador* como una clase anidada de clase *ControlDeCírculo*.
- Se registra el manejador para el botón *btAgrandar* y se implementa el método *handle()* en *AgrandarManejador* para invocar *panelCírculo.agrandar()*.
- Para reducir se hacen pasos similares de creación y registro del manejador correspondiente y la creación de la clase manejadora implementando la interfaz *EventHandler <ActionEvent >*.

El código de nuestro [ejemplo](#) quedaría como sigue:

```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Pos;
5  import javafx.scene.Scene;
6  import javafx.scene.control.Button;
7  import javafx.scene.layout.StackPane;
8  import javafx.scene.layout.HBox;
9  import javafx.scene.layout.BorderPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Circle;
12 import javafx.stage.Stage;
13
14 public class ControlDeCírculo extends Application {
15     private PanelCírculo panelCírculo = new PanelCírculo();
16
17     @Override
18     public void start(Stage escenarioPrimario) {
19
20         HBox hBox = new HBox();
```

```
21     hbox.setSpacing(10);
22     hbox.setAlignment(Pos.CENTER);
23     Button btAgrandar = new Button("Agrandar");
24     Button btReducir = new Button("Reducir");
25     hbox.getChildren().add(btAgrandar);
26     hbox.getChildren().add(btReducir);
27
28     // Crear y registrar el manejador para reducir
29     btReducir.setOnAction(new ReducirManejador());
30     // Crear y registrar el manejador para agrandar
31     btAgrandar.setOnAction(new AgrandarManejador());
32
33     BorderPane borderPanel = new BorderPane();
34     borderPanel.setCenter(panelCírculo);
35     borderPanel.setBottom(hbox);
36     BorderPane.setAlignment(hbox, Pos.CENTER);
37
38     // Crear una escena y colocarla en el escenario
39     Scene escena = new Scene(borderPanel, 200, 150);
40     escenarioPrimario.setTitle("ControlDeCírculo");
41     escenarioPrimario.setScene(escena);
42     escenarioPrimario.show();
43 }
44
45 class AgrandarManejador implements EventHandler<ActionEvent> {
46
47     @Override
48     public void handle(ActionEvent e) {
49         panelCírculo.agrandar();
50     }
51 }
52
53 class ReducirManejador implements EventHandler<ActionEvent> {
54
55     @Override
56     public void handle(ActionEvent e) {
57         panelCírculo.reducir();
58     }
59 }
60 }
61
62 class PanelCírculo extends StackPane {
```

```
63     private Circle círculo = new Circle(50);
64
65     public PanelCírculo() {
66         getChildren().add(círculo);
67         círculo.setStroke(Color.BLACK);
68         círculo.setFill(Color.WHITE);
69     }
70
71     public void agrandar() {
72         círculo.setRadius(círculo.getRadius() + 2);
73     }
74
75     public void reducir() {
76         círculo.setRadius(círculo.getRadius() > 2 ?
77             círculo.getRadius() - 2 : círculo.getRadius());
78     }
79 }
```

Listing 276. Ejemplo JavaFX dos botones con manejo de eventos.

33.3.1 Usando clases anónimas

Las clases anónimas son una muy buena opción para implementar una interfaz que contiene algunos métodos, ya que estas clase generalmente solo requieren de una instancia y no es necesario nombrar la clase. Ejemplo¹:

```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.layout.StackPane;
7  import javafx.stage.Stage;
8
9  public class ClaseAnónimaEjemplo extends Application {
10
11     @Override
12     public void start(Stage escenarioPrimario) {
13         escenarioPrimario.setTitle(";Hola clases anónimas!");
14         Button btn = new Button();
```

¹Anonymous classes

```

15     btn.setText("Decir 'Hola'");
16
17     //clase anónima dentro de setOnAction
18     btn.setOnAction(new EventHandler<ActionEvent>() {
19
20         @Override
21         public void handle(ActionEvent event) {
22             System.out.println(";Hola!");
23         }
24     });
25
26     StackPane raíz = new StackPane();
27     raíz.getChildren().add(btn);
28     escenarioPrimario.setScene(new Scene(raíz, 300, 250));
29     escenarioPrimario.show();
30 }
31 }

```

Listing 277. Ejemplo Java FX manejador de eventos con clase anónimas.

Otro ejemplo, en este caso el programa usa una clase anónima donde sobreescribe la implementación de clase *TextField* para los métodos *replaceText()* y *replaceSelection()*, creando un campo de texto que solo acepta valores numéricos:

```

1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.event.EventHandler;
4  import javafx.geometry.Insets;
5  import javafx.scene.Group;
6  import javafx.scene.Scene;
7  import javafx.scene.control.*;
8  import javafx.scene.layout.GridPane;
9  import javafx.scene.layout.HBox;
10 import javafx.stage.Stage;
11
12 public class ClaseAnónimaTextFieldAdaptado extends Application {
13
14     final static Label etiqueta = new Label();
15
16     @Override
17     public void start(Stage escenarioPrimario) {
18         Group raíz = new Group();

```

```
19     Scene escena = new Scene(raíz, 300, 150);
20     escenarioPrimario.setScene(escena);
21     escenarioPrimario.setTitle("Ejemplo de campo de texto");
22
23     GridPane cuadrícula = new GridPane();
24     // asigna las propiedades de relleno de espacio con un objeto de Insets
25     cuadrícula.setPadding(new Insets(10, 10, 10, 10));
26     cuadrícula.setVgap(5);
27     cuadrícula.setHgap(5);
28
29     escena.setRoot(cuadrícula);
30     final Label dinero = new Label("$");
31     GridPane.setConstraints(dinero, 0, 0);
32     cuadrícula.getChildren().add(dinero);
33
34     final TextField tfSum = new TextField() {
35         @Override
36         public void replaceText(int inicio, int fin, String texto) {
37             if (!texto.matches("[a-z, A-Z]")) {
38                 super.replaceText(inicio, fin, texto);
39             }
40             etiqueta.setText("Introduce un valor numérico");
41         }
42
43         @Override
44         public void replaceSelection(String texto) {
45             if (!texto.matches("[a-z, A-Z]")) {
46                 super.replaceSelection(texto);
47             }
48         }
49     };
50
51     tfSum.setPromptText("Introduce el total");
52     tfSum.setPrefColumnCount(10);
53     GridPane.setConstraints(tfSum, 1, 0);
54     cuadrícula.getChildren().add(tfSum);
55
56     Button btEnviar = new Button("Enviar");
57     GridPane.setConstraints(btEnviar, 2, 0);
58     cuadrícula.getChildren().add(btEnviar);
59
60     btEnviar.setOnAction(new EventHandler<ActionEvent>() {
```

```
61         @Override
62         public void handle(ActionEvent e) {
63             etiqueta.setText(null);
64         }
65     });
66
67     GridPane.setConstraints(etiqueta, 0, 1);
68     GridPane.setColumnSpan(etiqueta, 3);
69     cuadrícula.getChildren().add(etiqueta);
70
71     escena.setRoot(cuadrícula);
72     escenarioPrimario.show();
73 }
74
75 }
```

Listing 278. Ejemplo JavaFX con clases anónimas y uso de *TextField*.

33.3.2 Usando expresiones lambda

Las expresiones lambda introducidas en Java 8 son una forma de simplificar más el código para el manejo de eventos. Veamos un ejemplo de implementación de un manejador con expresiones lambda:

```
1  import javafx.application.Application;
2  import javafx.event.ActionEvent;
3  import javafx.geometry.Pos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.layout.HBox;
7  import javafx.stage.Stage;
8
9  public class EjemploManejadorLambda extends Application {
10     @Override
11     public void start(Stage escenarioPrimario) {
12
13         HBox hBox = new HBox();
14         hBox.setSpacing(10);
15         hBox.setAlignment(Pos.CENTER);
16         Button btNuevo = new Button("Nuevo");
17         Button btAbrir = new Button("Abrir");
18         Button btGrabar = new Button("Grabar");
```

```
19     Button btImprimir = new Button("Imprimir");
20     hbox.getChildren().addAll(btNuevo, btAbrir, btGrabar, btImprimir);
21
22     // Crear y registrar el manejador de cada botón con expresiones lambda
23     // Notar las diferentes formas es expresiones lambda
24     btNuevo.setOnAction((ActionEvent e) -> {
25         System.out.println("Proceso Nuevo");
26     });
27
28     btAbrir.setOnAction((e) -> {
29         System.out.println("Proceso Abrir");
30     });
31
32     btGrabar.setOnAction(e -> {
33         System.out.println("Proceso Grabar");
34     });
35
36     btImprimir.setOnAction(e -> System.out.println("Proceso Imprimir"));
37
38     Scene escena = new Scene(hbox, 300, 50);
39     escenarioPrimario.setTitle("Ejemplo de Manejador con Expresiones Lambda");
40     escenarioPrimario.setScene(escena);
41     escenarioPrimario.show();
42 }
43 }
```

Listing 279. Ejemplo JavaFX y manejador de eventos con expresiones lambda.

33.3.3 Ejemplo con evento de mouse

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.text.Text;
5  import javafx.stage.Stage;
6
7  public class EjemploEventoMouse extends Application {
8      @Override
9      public void start(Stage escenarioPrimario) {
10
11         Pane panel = new Pane();
```



```
12     Text texto = new Text(20, 20, "Presiona y arrastra un texto cualquiera");
13     panel.getChildren().addAll(texto);
14
15     texto.setOnMouseDragged(e -> {
16         texto.setX(e.getX());
17         texto.setY(e.getY());
18     });
19
20     Scene escena = new Scene(panel, 300, 100);
21     escenarioPrimario.setTitle("Ejemplo de Evento de Mouse");
22     escenarioPrimario.setScene(escena);
23     escenarioPrimario.show();
24 }
25 }
```

Listing 280. Ejemplo JavaFX con evento de mouse.

33.3.4 Ejemplo con evento de teclado

```
1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.text.Text;
5 import javafx.stage.Stage;
6
7 public class EjemploEventoTeclado extends Application {
8     @Override
9     public void start(Stage escenarioPrimario) {
10
11         Pane panel = new Pane();
12         panel.setMinSize(300, 300);
13         Text texto = new Text(20, 20, "A");
14         panel.getChildren().add(texto);
15         texto.setOnKeyPressed(e -> {
16             switch (e.getCode()) {
17                 case DOWN: texto.setY(texto.getY() + 10); break;
18                 case UP: texto.setY(texto.getY() - 10); break;
19                 case LEFT: texto.setX(texto.getX() - 10); break;
20                 case RIGHT: texto.setX(texto.getX() + 10); break;
21                 default:
22                     if (Character.isLetterOrDigit(e.getText().charAt(0)))
```

```
23         texto.setText(e.getText());
24     }
25 });
26
27     Scene escena = new Scene(panel);
28     escenarioPrimario.setTitle("Ejemplo Evento Teclado");
29     escenarioPrimario.setScene(escena);
30     escenarioPrimario.show();
31     texto.requestFocus(); // texto se enfoca para recibir la entrada de teclado
32 }
33 }
```

Listing 281. Ejemplo JavaFX con evento de teclado.

33.3.5 Ejemplo con evento de teclado y mouse

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Button;
5  import javafx.scene.input.KeyCode;
6  import javafx.scene.input.MouseButton;
7  import javafx.scene.layout.HBox;
8  import javafx.scene.layout.BorderPane;
9  import javafx.scene.layout.StackPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Circle;
12 import javafx.stage.Stage;
13
14 public class EjemploEventoTecladoMouse extends Application {
15     private PanelCírculo panelCírculo = new PanelCírculo();
16
17     @Override
18     public void start(Stage escenarioPrimario) {
19         // mantener dos botones en un HBox
20         HBox hBox = new HBox();
21         hBox.setSpacing(10);
22         hBox.setAlignment(Pos.CENTER);
23         Button btAgrandar = new Button("Agrandar");
24         Button btReducir = new Button("Reducir");
25         hBox.getChildren().add(btAgrandar);
```

```
26     hbox.getChildren().add(btReducir);
27
28     // Crear y registrar manejadores
29     btAgrandar.setOnAction(e -> panelCírculo.agrandar());
30     btReducir.setOnAction(e -> panelCírculo.reducir());
31
32     panelCírculo.setOnMouseClicked(e -> {
33         if (e.getButton() == MouseButton.PRIMARY) {
34             panelCírculo.agrandar();
35         } else if (e.getButton() == MouseButton.SECONDARY) {
36             panelCírculo.reducir();
37         }
38     });
39
40     panelCírculo.setOnKeyPressed(e -> {
41         if (e.getCode() == KeyCode.U) {
42             panelCírculo.agrandar();
43         } else if (e.getCode() == KeyCode.D) {
44             panelCírculo.reducir();
45         }
46     });
47
48     BorderPane panelBorde = new BorderPane();
49     panelBorde.setCenter(panelCírculo);
50     panelBorde.setBottom(hbox);
51     BorderPane.setAlignment(hbox, Pos.CENTER);
52
53     Scene escena = new Scene(panelBorde, 200, 150);
54     escenarioPrimario.setTitle("EjemploEventoTecladoMouse");
55     escenarioPrimario.setScene(escena);
56     escenarioPrimario.show();
57     panelCírculo.requestFocus();
58 }
59 }
60
61 // vista en ejemplo anterior
62 class PanelCírculo extends StackPane {
63     private Circle círculo = new Circle(50);
64
65     public PanelCírculo() {
66         getChildren().add(círculo);
67         círculo.setStroke(Color.BLACK);
```

```
68     círculo.setFill(Color.WHITE);
69 }
70
71 public void agrandar() {
72     círculo.setRadius(círculo.getRadius() + 2);
73 }
74
75 public void reducir() {
76     círculo.setRadius(círculo.getRadius() > 2 ?
77         círculo.getRadius() - 2 : círculo.getRadius());
78 }
79 }
```

Listing 282. Ejemplo JavaFX con evento de teclado y mouse.

33.3.6 Ejemplo con listener en un objeto observable

Es posible añadir un *listener* para procesar un cambio en un valor en un objeto observable. Una instancia de *Observable* es conocida como un **objeto observable**.

Ejemplo:

```
1  import javafx.beans.InvalidationListener;
2  import javafx.beans.Observable;
3  import javafx.beans.property.DoubleProperty;
4  import javafx.beans.property.SimpleDoubleProperty;
5
6  public class EjemploPropiedadObservable {
7      public static void main(String[] args) {
8          DoubleProperty balance = new SimpleDoubleProperty();
9          balance.addListener(new InvalidationListener() {
10              @Override
11              public void invalidated(Observable ov) {
12                  System.out.println("El nuevo valor es " + balance.doubleValue());
13              }
14          });
15          balance.set(4.5);
16      }
17  }
```

Listing 283. Ejemplo JavaFX con un objeto observable.

Capítulo 34

Programación en Red con Java

Java, como un lenguaje de programación moderno, provee de clases para el manejo de información en red. De hecho, el uso de otras tecnologías como JDBC involucra el acceso a bases de datos locales y remotas de forma prácticamente transparente.

34.1 Paquete *java.net*

Dentro del paquete *java.net* se tienen un conjunto de clases que dan soporte a los diversos protocolos de comunicación de Internet, conocido como paquete de protocolos de Internet, dentro de los cuales se encuentran:

1. IP. *Internet Protocol*.
2. TCP. *Transmission Control Protocol*.
3. UDP. *User Datagram Protocol*.

La mayor parte de las aplicaciones están basadas en los protocolos TCP/IP, las cuales muchas veces hacen uso de otros protocolos intermediarios entre TCP/IP y la aplicación.

La tabla ?? muestra una lista de protocolos de uso común en Internet.

Cuadro 34.1. Protocolos comunes

Acrónimo	Nombre	Descripción
HTTP	<i>HyperText Transport Protocol</i>	Protocolo de hipertexto. Es la base del World Wide Web.
FTP	<i>File Transfer, Protocol</i>	Protocolo de transferencia de archivos.
POP3	<i>Post Office Protocol</i>	Protocolo que permite el acceso al correo electrónico.
SMTP	<i>Simple Mail Transfer Protocol</i>	Protocolo para transferencia de correo electrónico.
NNTP	<i>Network News Transfer Protocol</i>	Protocolo para grupos de noticias (news)

El paquete *java.net* cuenta hasta la versión 1.4 del jdk con 6 interfaces, 27 clases y 11 excepciones. Las clases más usadas son:

1. *URL*. Representa un URL de Internet.
2. *URLConnection*. Es un complemento de URL (no una subclase de ella). Cubre algunas operaciones más complejas.
3. *Socket*. Establece conexiones TCP/IP.
4. *DatagramPacket*. Establece conexiones de tipo UDP.
5. *InetAddress*. Representa una dirección IP.

34.1.1 Clase *URL*

Esta clase permite crear instancias que almacenen direcciones de recursos en Internet¹. Este recurso puede ser un archivo, directorio, o inclusive una consulta a un motor de búsqueda. En caso de que el URL tenga una sintaxis incorrecta se lanza una excepción *MalformedURLException*.

Ejemplo:

```
1  //Ejemplo de uso de URL con JavaFX
2  import javafx.application.Application;
3  import javafx.stage.Stage;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.layout.StackPane;
7
8  public class URLEjemplo extends Application {
9
10     @Override
11     public void start(Stage primaryStage) {
12         Button btn = new Button("Ir a www.utm.mx");
13         btn.setOnAction(e -> {
14             getHostServices().showDocument("http://www.utm.mx");
15         });
16
17         StackPane root = new StackPane();
18         root.getChildren().add(btn);
19
20         Scene scene = new Scene(root, 300, 250);
```

¹URL. Uniform Resource Locator

```
21
22     primaryStage.setTitle("Ejemplo URL");
23     primaryStage.setScene(scene);
24     primaryStage.show();
25 }
26
27 public static void main(String[] args) {
28     launch(args);
29 }
30 }
```

Listing 284. Ejemplo de clase URL con JavaFX.

Este programa muestra una ventana con un botón que, al ser presionado, abre la página web indicada en el navegador predeterminado del sistema.

Este es sólo un ejemplo, pero la clase URL es usada por todos aquellos programas que requieran del uso de direcciones de recursos de Internet.

34.1.2 Clase *InetAddress*

Como se mencionó antes, esta clase representar una dirección IP. La clase no maneja atributos ni constructores. Ofrece en cambio métodos de acceso para operaciones comunes de Internet.

Ejemplo:

```
1 //obtiene la dirección IP de la máquina local
2
3 import java.net.*;
4
5 public class ObtenIPLocal {
6
7     public static void main(String args[]) {
8         InetAddress IPLocal=null;
9
10        try {
11            IPLocal= InetAddress.getLocalHost();
12        } catch (UnknownHostException e) {}
13
14        System.out.println(IPLocal);
15    }
16 }
```

Listing 285. Ejemplo de *InetAddress*, obtiene la dirección IP de la máquina local.

El programa anterior usa una instancia de *InetAddress* para obtener la dirección de la máquina local mediante el método *getLocalHost()* de la clase. Es un ejemplo muy simple del uso de la clase.

El siguiente programa recibe como parámetro el nombre de un servidor y obtiene la dirección asociada a ese nombre.

Ejemplo:

```
1 //identifica la direccion IP asociada al host
2 import java.net.InetAddress;
3 import java.net.UnknownHostException;
4 import java.lang.System;
5
6 public class NSLookupApp {
7     public static void main(String args[]) {
8         try {
9             if(args.length!=1) {
10                 System.out.println("Sintaxis: java NSLookupApp nombreServidor");
11                 return;
12             }
13             InetAddress host = InetAddress.getBy_name(args[0]);
14             String hostName = host.getHostName();
15             System.out.println("Nombre servidor: "+hostName);
16             System.out.println("Direccion IP: "+host.getHostAddress());
17         } catch (UnknownHostException ex) {
18             System.out.println("Servidor desconocido");
19             return;
20         }
21     }
22 }
```

Listing 286. Ejemplo de *InetAddress*, identifica la dirección IP asociada al *host*. Ejecutando por ejemplo:

```
\$java NSLookupApp www.utm.mx
```

34.1.3 Clase *Socket*

Esta clase se usa para la implementación de *sockets* de cliente basados en una conexión. La aplicación cliente debe comúnmente iniciar la conexión de *sockets* hacia el servidor.

Una instancia de la clase *Socket* es creada con el constructor recibiendo como parámetros por lo general el número IP o nombre de dominio del servidor y el puerto del servidor, creando una conexión a un puerto y *host* de destino.

Un *socket* se puede crear de la siguiente forma:

```
miSocket = new Socket ("mixteco.utm.mx", 1111);
```

Esta línea de código tiene que estar dentro de un segmento *try* para recibir una excepción en caso de que se produzca.

Algunos métodos importantes de la clase *Socket*:

- *getInetAddress()*. Obtiene la dirección IP del servidor destino.
- *getPort()*. Obtiene el puerto del servidor destino.
- *getLocalAddress()*. Obtiene la dirección IP local.
- *getLocalPort()*. Obtiene el número de puerto local.
- *getInputStream()*. Para acceder a los flujos de entrada.
- *getOutputStream()*. Para acceder a los flujos de salida.
- *close()*. Cerrar el socket cliente.

34.1.4 Clase *ServerSocket*

Esta clase implementa un *socket* del servidor TCP. Una instancia de la clase recibe comúnmente el número de puerto por el cual va a **escuchar** las solicitudes de conexión del cliente.

Dentro de código para manejo de excepciones se declara un *socket* servidor de la siguiente forma:

```
miServidor = new ServerSocket (1111);
```

Algunos métodos importantes de la clase *ServerSocket*:

- *accept()*. Hace que el *socket* servidor escuche y espere hasta que se establezca una conexión entrante.
- *getSoTimeout()*. Devuelve el tiempo que va a estar bloqueado el *socket* con respecto a una llamada al método *accept()*.
- *setSoTimeout()*. Modifica el tiempo de bloqueo del *socket*.
- *close()*. Cierra el *socket* servidor.

Veamos ahora un ejemplo con una clase cliente y otra clase servidor. Este programa aprovecha las características de multihilos de Java creando un hilo cliente y otro servidor.

[Ejemplo:](#)

```
1  //Programa cliente servidor con sockets
2  import java.awt.*;
3  import java.net.*;
4  import java.io.*;
5
6  class hiloCliente extends Thread {
7
8      BufferedReader dis = null;
9      Socket s = null;
10
11     public hiloCliente() {
12         try {
13             //se crea socket con dirección de máquina local
14             s = new Socket("127.0.0.1", 2525);
15             dis = new BufferedReader(new InputStreamReader(s.getInputStream()));
16         }
17         catch (IOException e)
18         {
19             System.out.println("Error: " + e);
20         }
21     }
22
23     public void run()
24     {
25         while (true)
26         {
27             try {
28                 String mensaje = dis.readLine();
29                 if (mensaje == null)
30                     break;
31                 System.out.println(mensaje);
32             }
33             catch (IOException e)
34             {
35                 System.out.println("Error: " + e);
36             }
37         }
38     }
39 }
40
41 public class clienteYServidor extends Frame {
```

```
42  static ServerSocket servidor = null;
43
44  public boolean handleEvent (Event evt)
45  {
46      if (evt.id == Event.WINDOW_DESTROY)
47      {
48          System.exit(0);
49      }
50      return super.handleEvent (evt);
51  }
52
53  public boolean mouseDown(Event evt, int x, int y)
54  {
55      new hiloCliente().start(); // Iniciar el hilo cliente
56      return(true);
57  }
58
59  public static void main(String args[])
60  {
61      clienteYServidor f = new clienteYServidor();
62      f.resize (200, 200);
63      f.show();
64      try {
65          //genera el socket servidor
66          servidor = new ServerSocket(2525);
67      }
68      catch (IOException e)
69      {
70          System.out.println("Error: " + e);
71      }
72      while (true)
73      {
74          Socket s = null;
75          try {
76              s = servidor.accept();
77          }
78          catch (IOException e)
79          {
80              System.out.println("Error: " + e);
81          }
82
83          try {
```

```

84         PrintStream ps = new PrintStream(s.getOutputStream());
85         ps.println("Hola, Mundo");
86         ps.flush();
87         s.close();
88     }
89     catch (IOException e)
90     {
91         System.out.println("Error: " + e);
92     }
93 }
94 }
95 }

```

Listing 287. Ejemplo de programa cliente servidor con sockets .

El programa muestra a un hilo cliente solicitando una cadena de un flujo de datos al servidor: cada vez que se da *click* sobre la ventana el servidor inicia a un hilo cliente que a su vez se comunica con el servidor.

En el **sección complementaria uno** se muestra otro ejemplo de uso de la clase *Socket* para crear un cliente y en la **sección complementaria dos** se muestra el código correspondiente al servidor. Estos programas se comunican entre sí: el cliente es capaz de enviar una cadena y recibirla de vuelta modificada por el servidor. Cada uno corre de manera independiente e idealmente debería ser probado en distintas máquinas en red.

Ejemplo de ejecución del **cliente**:

```

1 $ java PortTalkApp yodocono.utm.mx 1234
2 Conectando a: yodocono.utm.mx puerto 1234.
3 servidor destino yodocono.utm.mx.
4 IP servidor destino 192.100.170.5.
5 numero de puerto servidor destino 1234.
6 servidor Local iec23.
7 IP servidor Local 192.100.170.33.
8 numero de puerto servidor Local 2162.
9 Enviar, recibir, o salir (E/R/S): e
10 Hola yodocono
11 Enviar, recibir, o salir (E/R/S): r
12 ***onocodoy aloH

```

Ejemplo de ejecución del **servidor**:

```

1 $ java ReverServerApp

```

```
2 Servidor escuchando en puerto: 1234.
3 Aceptando conexion a iec23.utm.mx en puerto 2162.
4 Recibido: Hola yodocono
5 Enviado: onocodoy aloH
```

34.1.5 Clase *DatagramSocket*

Esta clase es el punto de entrada de todas las acciones sobre datagramas UDP². Sería el equivalente a la clase *Socket* y *ServerSocket* para el protocolo TCP, ya que implementa los *sockets* cliente y servidor.

Principales métodos³ de la clase *DatagramSocket*:

- *send()*. Enviar un datagrama a través del *socket*.
- *receive()*. Recibir un datagrama a través del *socket*.
- *close()*. Cerrar el *socket*.

34.1.6 Clase *DatagramPacket*

Esta clase representa un paquete de datos recibido o enviado mediante un *socket* a través del protocolo UDP. Se le considera una clase de bajo nivel que sólo resulta útil para aplicaciones que deben leer o escribir datos según un formato específico, pero que no necesitan garantizar la integridad de la llamada.

Cada datagrama es enviado de una máquina a otra a partir de la información del paquete. Si un conjunto de paquetes son enviados hacia una máquina estos podrían ser enviados por caminos distintos y tener un orden de llegada distinto.

A continuación se muestra la salida generada por dos programas que se detallan en las **secciones complementarias tres y cuatro**. El programa *TimeServerApp* esta a la espera de solicitudes de **tiempo** o **salida** y de acuerdo a estas solicitudes enviará al cliente la fecha y hora o provocará la finalización del servidor.

Ejemplo de ejecución de *TimeServerApp*:

```
1 $ java TimeServerApp
2 iec23: TimeServer escuchando el puerto 2345.
3
4 Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
5 Contenido del datagrama: tiempo
```

²UDP es un protocolo que carece de conexión y que permite que los programas de aplicación intercambien información por medio de trozos de información a los que se conoce datagramas.

³Algunos métodos importantes no se mencionan porque son comunes a los proporcionados por otras clases con anterioridad.

```

6 Enviando: Fri Jun 02 17:20:58 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.
7
8 Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
9 Contenido del datagrama: tiempo
10 Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.
11
12 Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
13 Contenido del datagrama: tiempo
14 Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.
15
16 Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
17 Contenido del datagrama: tiempo
18 Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.
19
20 Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
21 Contenido del datagrama: tiempo
22 Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.
23
24 Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
25 Contenido del datagrama: salida
26 Enviando: Fri Jun 02 17:21:00 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

```

Por su parte, el programa *GetTimeApp* envía cinco solicitudes de "tiempoz una de "salida." al servidor.

Ejemplo de ejecución de *GetTimeApp*:

```

1 $ java GetTimeApp
2
3 Envía petición de tiempo a iec23 al puerto 2345.
4 Recibiendo un datagrama desde iec23.utm.mx at port 2345.
5 El datagrama contiene los sig. datos: Fri Jun 02 17:20:58 GMT-05:00 2000
6
7 Envía petición de tiempo a iec23 al puerto 2345.
8 Recibiendo un datagrama desde iec23.utm.mx at port 2345.
9 El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000
10
11 Envía petición de tiempo a iec23 al puerto 2345.
12 Recibiendo un datagrama desde iec23.utm.mx at port 2345.
13 El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000
14
15 Envía petición de tiempo a iec23 al puerto 2345.
16 Recibiendo un datagrama desde iec23.utm.mx at port 2345.

```

```

17 El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000
18
19 Envía petición de tiempo a iec23 al puerto 2345.
20 Recibiendo un datagrama desde iec23.utm.mx at port 2345.
21 El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

```

En este ejemplo se asume el funcionamiento en una sola máquina pues se toma la dirección de la máquina local, pero modificarlo para probarlo en máquinas distintas no debe ser problema.

34.2 Complemento 1. *PortTalkApp*

```

1  //Ejemplo de uso de la clase Socket
2  import java.lang.System;
3  import java.net.Socket;
4  import java.net.InetAddress;
5  import java.net.UnknownHostException;
6  import java.io.*;
7
8  public class PortTalkApp {
9      public static void main(String args[]){
10         PortTalk portTalk = new PortTalk(args);
11         portTalk.displayDestinationParameters();
12         portTalk.displayLocalParameters();
13         portTalk.chat();
14         portTalk.shutdown();
15     }
16 }
17
18 class PortTalk {
19     Socket connection;
20     DataOutputStream outStream;
21     BufferedReader inStream;
22     public PortTalk(String args[]){
23         if(args.length!=2) error("Sintaxis: java PortTalkApp <servidor> <puerto>");
24         String destination = args[0];
25         int port = 0;
26         try {
27             port = Integer.valueOf(args[1]).intValue();
28         } catch (NumberFormatException ex){

```

```
29     error("Número de puerto inv lido");
30 }
31 try{
32     connection = new Socket(destination,port);
33 } catch (UnknownHostException ex){
34     error("Servidor desconocido");
35 }
36 catch (IOException ex){
37     error("Error E/S: al crear el socket");
38 }
39 try{
40     inStream = new BufferedReader(
41         new InputStreamReader(connection.getInputStream()));
42     outStream = new DataOutputStream(connection.getOutputStream());
43 } catch (IOException ex){
44     error("Error E/S: obteniendo el flujo");
45 }
46 System.out.println("Conectando a: "+destination+" puerto "+port+".");
47 }
48 public void displayDestinationParameters() {
49     InetAddress destAddress = connection.getInetAddress();
50     String name = destAddress.getHostName();
51     byte ipAddress[] = destAddress.getAddress();
52     int port = connection.getPort();
53     displayParameters("servidor destino ", name, ipAddress, port);
54 }
55 public void displayLocalParameters() {
56     InetAddress localAddress = null;
57     try{
58         localAddress = InetAddress.getLocalHost();
59     } catch (UnknownHostException ex){
60         error("Error obteniendo informaci n de servidor local");
61     }
62     String name = localAddress.getHostName();
63     byte ipAddress[] = localAddress.getAddress();
64     int port = connection.getLocalPort();
65     displayParameters("servidor Local ", name, ipAddress, port);
66 }
67 public void displayParameters(String s, String name, byte ipAddress[], int port) {
68     System.out.println(s+name+".");
69     System.out.print("IP "+s);
70     for(int i=0; i<ipAddress.length; ++i)
```



```
71     System.out.print((ipAddress[i]+256)%256+".");
72     System.out.println();
73     System.out.println("numero de puerto "+s+port+".");
74 }
75 public void chat() {
76     BufferedReader keyboardInput = new BufferedReader(
77         new InputStreamReader(System.in));
78     boolean finished = false;
79     do {
80         try{
81             System.out.print("Enviar, recibir, o salir (E/R/S): ");
82             System.out.flush();
83             String line = keyboardInput.readLine();
84             if(line.length()>0){
85                 line=line.toUpperCase();
86                 switch (line.charAt(0)){
87                     case 'E':
88                         String sendLine = keyboardInput.readLine();
89                         outputStream.writeBytes(sendLine);
90                         outputStream.write(13);
91                         outputStream.write(10);
92                         outputStream.flush();
93                         break;
94                     case 'R':
95                         int inByte;
96                         System.out.print("***");
97                         while ((inByte = inputStream.read()) != '\n')
98                             System.out.write(inByte);
99                         System.out.println();
100                        break;
101                     case 'S':
102                         finished=true;
103                         break;
104                     default:
105                         break;
106                 }
107             }
108         } catch (IOException ex){
109             error("Error leyendo del teclado o socket");
110         }
111     } while(!finished);
112 }
```

```
113 public void shutdown() {
114     try{
115         connection.close();
116     } catch (IOException ex) {
117         error("Error e/S cerrando socket");
118     }
119 }
120 public void error(String s) {
121     System.out.println(s);
122     System.exit(1);
123 }
124 }
```

Listing 288. Ejemplo de uso de la clase Socket. *PortTalkApp*.

34.3 Complemento 2. ServerSocket

```
1 //ejemplo de uso de la clase ServerSocket
2 import java.lang.System;
3 import java.net.ServerSocket;
4 import java.net.Socket;
5 import java.io.*;
6
7 public class ReverServerApp {
8     public static void main(String args[]) {
9         try{
10             ServerSocket server = new ServerSocket(1234);
11             int localPort = server.getLocalPort();
12             System.out.println("Servidor escuchando en puerto: "+localPort+".");
13             Socket client = server.accept();
14             String destName = client.getInetAddress().getHostName();
15             int destPort = client.getPort();
16             System.out.println("Aceptando conexión a "+destName+" en puerto "+
17                 destPort+".");
18             BufferedReader inStream = new BufferedReader(
19                 new InputStreamReader(client.getInputStream()));
20             DataOutputStream outStream = new DataOutputStream(client.getOutputStream());
21             boolean finished = false;
22             do {
23                 String inLine = inStream.readLine();
```

```

24     System.out.println("Recibido: "+inLine);
25     if(inLine.equalsIgnoreCase("salir")) finished=true;
26     String outLine=new ReverseString(inLine.trim()).getString();
27     for(int i=0;i<outLine.length();++i)
28         outputStream.write((byte)outLine.charAt(i));
29     outputStream.write(13);
30     outputStream.write(10);
31     outputStream.flush();
32     System.out.println("Enviado: "+outLine);
33 } while(!finished);
34 inStream.close();
35 outputStream.close();
36 client.close();
37 server.close();
38 } catch (IOException ex){
39     System.out.println("excepcion: IOException .");
40 }
41 }
42 }
43 class ReverseString {
44     String s;
45     public ReverseString(String in){
46         int len = in.length();
47         char outChars[] = new char[len];
48         for(int i=0;i<len;++i)
49             outChars[len-1-i]=in.charAt(i);
50         s = String.valueOf(outChars);
51     }
52     public String getString(){
53         return s;
54     }
55 }

```

Listing 289. Ejemplo de uso de la clase *ServerSocket*.

34.4 Complemento 3. *TimeServerApp*

```

1 //ejemplo de escucha de un socket UDP
2 import java.lang.System;
3 import java.net.DatagramSocket;

```

```
4 import java.net.DatagramPacket;
5 import java.net.InetAddress;
6 import java.io.IOException;
7 import java.util.Date;
8
9 public class TimeServerApp {
10     public static void main(String args[]) {
11         try{
12             DatagramSocket socket = new DatagramSocket(2345);
13             String localAddress = InetAddress.getLocalHost().getHostName().trim();
14             int localPort = socket.getLocalPort();
15             System.out.print(localAddress+": ");
16             System.out.println("TimeServer escuchando el puerto "+localPort+".");
17             int bufferLength = 256;
18             byte outBuffer[];
19             byte inBuffer[] = new byte[bufferLength];
20             DatagramPacket outDatagram;
21             DatagramPacket inDatagram = new DatagramPacket(inBuffer,inBuffer.length);
22             boolean finished = false;
23             do {
24                 socket.receive(inDatagram);
25                 InetAddress destAddress = inDatagram.getAddress();
26                 String destHost = destAddress.getHostName().trim();
27                 int destPort = inDatagram.getPort();
28                 System.out.println("\nRecibiendo un datagrama desde "+destHost+" puerto "+
29                     destPort+".");
30                 String data = new String(inDatagram.getData()).trim();
31                 System.out.println("Contenido del datagrama: "+data);
32                 if(data.equalsIgnoreCase("salida")) finished=true;
33                 String time = new Date().toString();
34                 outBuffer=time.getBytes();
35                 outDatagram = new DatagramPacket(outBuffer,outBuffer.length,destAddress,
36                     destPort);
37                 socket.send(outDatagram);
38                 System.out.println("Enviando: "+time+" a "+destHost+" al puerto "+destPort+".");
39             } while(!finished);
40         } catch (IOException ex){
41             System.out.println("Excepcion: IOException");
42         }
43     }
44 }
```

45

Listing 290. Ejemplo de escucha de un socket UDP. *TimeServerApp*

34.5 Complemento 4. *GetTimeApp*

```
1 //ejemplo de uso de datagramas
2 import java.lang.System;
3 import java.net.DatagramSocket;
4 import java.net.DatagramPacket;
5 import java.net.InetAddress;
6 import java.io.IOException;
7
8 public class GetTimeApp {
9     public static void main(String args[]){
10         try{
11             DatagramSocket socket = new DatagramSocket();
12             InetAddress localAddress = InetAddress.getLocalHost();
13             String localHost = localAddress.getHostName();
14             int bufferSize = 256;
15             byte outBuffer[];
16             byte inBuffer[] = new byte[bufferLength];
17             DatagramPacket outDatagram;
18             DatagramPacket inDatagram = new DatagramPacket(inBuffer,inBuffer.length);
19             for(int i=0;i<5;++i){
20                 outBuffer = new byte[bufferLength];
21                 outBuffer = "tiempo".getBytes();
22                 outDatagram = new DatagramPacket(outBuffer,outBuffer.length,
23                     localAddress,2345);
24                 socket.send(outDatagram);
25                 System.out.println("\nEnvia peticion de tiempo a "+localHost+" al puerto 2345.");
26                 socket.receive(inDatagram);
27                 InetAddress destAddress = inDatagram.getAddress();
28                 String destHost = destAddress.getHostName().trim();
29                 int destPort = inDatagram.getPort();
30                 System.out.println("Recibiendo un datagrama desde "+destHost+" at port "+
31                     destPort+".");
32                 String data = new String(inDatagram.getData());
33                 data=data.trim();
34                 System.out.println("El datagrama contiene los sig. datos: "+data);
```

```
35     }
36     outBuffer = new byte[bufferLength];
37     outBuffer = "salida".getBytes();
38     outDatagram = new DatagramPacket(outBuffer, outBuffer.length,
39         localAddress, 2345);
40     socket.send(outDatagram);
41 } catch (IOException ex) {
42     System.out.println("excepci3n: IOException");
43 }
44 }
45 }
```

Listing 291. Ejemplo de uso de datagramas. *GetTimeApp*.

Ejercicios sugeridos

- • ¿Qué pasa si al programa *ReverServerApp* se tratan de conectar dos o más clientes? Proponga e implemente una solución para recibir más de un cliente.
- • Modifique el ejemplo de uso de datagramas para poder conectarse desde cualquier máquina en la red al servidor *TimeServerApp*.

Capítulo 35

Programación en Red con Java (2)

Una vez analizados los conceptos básicos de programación en red en el material pasado, revisaremos unas clases complementarias y veremos algunos ejemplos de aplicaciones cliente/servidor.

35.1 Clases *URL*

En el documento anterior se han revisado varias clases para la comunicación en red a través de TCP y de UDP. Sin embargo, Java proporciona otras clases de alto nivel y se encuentran organizadas alrededor de la clase *URL*.

Existen varias clases para el manejo de *URL* además de la propia clase *URL* vista anteriormente.

1. *URLConnection*. Es una clase abstracta, ofrece una conexión activa a un recurso representado en una instancia de *URL*. Tiene como subclases a *HttpURLConnection* y *JarURLConnection*. Proporciona la funcionalidad básica para que las instancias de sus subclases puedan leer o escribir del recurso apuntado por un *URL*.
2. *HttpURLConnection*. Extiende la clase *URLConnection*. Una instancia de esta clase permite la conexión a un servidor http.
3. *JarURLConnection*. Es usada para hacer referencia a un archivo jar o a un recurso contenido dentro de un archivo jar. La conexión sigue siendo bajo http. Sintaxis general:
jar :< url >!/entry
4. *URLEncoder*. Esta clase contiene un método estático para convertir una cadena en formato *x-www-form-urlencoded*¹.

¹Este formato es posible apreciarlo en el uso de CGI's; donde por ejemplo, un espacio es representado con el símbolo +.

5. *URLDecoder*. Al contrario de la clase anterior, una instancia de esta clase convierte del formato *x-www-form-urlencoded* a cadena.

35.2 Cliente / Servidor

Mucho se ha hablado de la tecnología cliente / servidor y como ésta es lograda en diferentes niveles. Por ejemplo, el uso de la JDBC es una arquitectura de este estilo donde la aplicación carga con las capas de manipulación y presentación de datos y el manejador de la base de datos obviamente tiene la responsabilidad del almacenamiento. Con la programación en red nosotros podemos realizar aplicaciones cliente y aplicaciones servidor, donde dependiendo de nuestras necesidades podamos proponer inclusive el movimiento de estas capas como se mencionaba en el curso propedéutico de introducción a la tecnología de objetos.

Retomando la JDBC, uno de los problemas del acceso usando el puente JDBC-ODBC es que debe estar configurado el acceso a la base de datos en cada máquina, lo que no es recomendable para cierto tipo de aplicaciones. Si no se tiene otra forma de acceder a la base de datos es posible hacer un cliente que solo se encargue de recibir la información y presentarla, y una aplicación del lado del servidor que acceda a la base de datos a través de JDBC-ODBC y envíe la información a través de conexiones de *sockets*.

De una manera más formal, un cliente y servidor se definen como sigue:

Cliente. Un cliente es una aplicación que realiza un servicio al usuario apoyado en tareas realizadas por un servidor, a través de solicitudes de servicio. Se asume que por lo general el cliente es quien contacta al servidor para realizar la conexión.

Servidor. Un servidor por su parte es una aplicación que se encuentra a la espera de conexiones de clientes a través de un puerto asociado a su servicio. Un servidor debe generar un hilo por cada cliente que se encuentre solicitando un servicio.

35.2.1 Programas cliente

Se muestran a continuación ejemplos de programas cliente básicos.

Ejemplo:

```
1 //Programa que almacena páginas html
2 import java.util.ArrayList;
3 import java.io.*;
4 import java.net.*;
5
6 public class CapturaHtmlApp {
7     public static void main(String args[]){
```



```
8   CapturaHtml captu = new CapturaHtml();
9   captu.run();
10  }
11  }
12
13  class CapturaHtml {
14      String urlList = "urlList.txt";
15      ArrayList<String> URLs = new ArrayList<>();
16      ArrayList<String> fileNames = new ArrayList<>();
17      public CapturaHtml() {
18          super();
19      }
20      public void getURLList() {
21          try {
22              BufferedReader inStream = new BufferedReader(new FileReader(urlList));
23              String inLine;
24              while((inLine = inStream.readLine()) != null) {
25                  inLine = inLine.trim();
26                  if(!inLine.equals("")) {
27                      int tabPos = inLine.lastIndexOf('\t');
28                      String url = inLine.substring(0,tabPos).trim();
29                      String fileName = inLine.substring(tabPos+1).trim();
30                      URLs.add(url);
31                      fileNames.add(fileName);
32                  }
33              }
34          } catch(IOException ex){
35              error("Error leyendo "+urlList);
36          }
37      }
38      public void run() {
39          getURLList();
40          int numURLs = URLs.size();
41          for(int i=0;i<numURLs;++i)
42              captuURL(URLs.get(i), fileNames.get(i));
43          System.out.println("Ok.");
44      }
45      public void captuURL(String urlName,String fileName) {
46          try{
47              URL url = new URL(urlName);
48              System.out.println("Obteniendo "+urlName+"...");
49              File outFile = new File(fileName);
```

```
50     PrintWriter outputStream = new PrintWriter(new FileWriter(outFile));
51     BufferedReader inputStream = new BufferedReader(
52         new InputStreamReader(url.openStream()));
53     String line;
54     while ((line = inputStream.readLine()) != null) outputStream.println(line);
55     inputStream.close();
56     outputStream.close();
57 } catch (MalformedURLException ex) {
58     System.out.println("MalformedURLException");
59 } catch (IOException ex) {
60     System.out.println("IOException");
61 }
62 }
63 public void error(String s) {
64     System.out.println(s);
65     System.exit(1);
66 }
67 }
```

Listing 292. Ejemplo que almacena páginas html.

Este programa toma un archivo `urlList.txt` para obtener una lista de direcciones de Web y almacenar localmente los archivos html resultantes, obtenidos de los servidores correspondientes. El archivo de direcciones puede ser:

```
http://www.utm.mx    utm.htm
http://virtual.utm.mx    virtual.htm
```

donde la dirección se encuentra separada del nombre del archivo local por un carácter de tabulación.

En el siguiente programa podemos ver a un cliente de SMTP para envío de correo, el cual manda un correo electrónico de prueba conectándose a un servidor SMTP. Para que se ejecute correctamente es necesario ajustar las cuentas de origen y destino, así como el nombre de dominio del servidor de SMTP.

Es posible que este programa no funcione con algunos servidores. Una de las razones puede ser por cuestiones de seguridad, ya que fácilmente se podría mandar un mensaje aparentando un remitente que no nos pertenece pues, como es posible apreciar, en el código no hay ninguna medida de seguridad de para comprobar nuestra identidad. Sin embargo muchos servidores SMTP no están activados para verificar la identidad del cliente².

Ejemplo:

²El código muestra servidores de ejemplo. En la fecha de prueba de este programa dicho servidor no verificaba la identidad del cliente pero esto pudo haber cambiado a la fecha.

```
1  //Ejemplo de cliente de SMTP
2  import java.io.*;
3  import java.net.*;
4
5  public class CorreoJava {
6      static PrintStream ps = null;          // envío de mensajes
7      static BufferedReader dis = null;
8
9
10     public static void enviar(String str) throws IOException
11     {
12         ps.println(str); // enviar una cadena SMTP
13         ps.flush();      // vaciar la cadena
14
15         System.out.println("Java envió: " + str);
16     }
17
18     public static void recibir() throws IOException
19     {
20         String readstr = dis.readLine(); // obtener la respuesta SMTP
21         System.out.println("respuesta SMTP: " + readstr);
22     }
23
24     public static void main (String args[])
25     {
26         String HELO = "HELO ";
27         String MAIL_FROM = "MAIL FROM: miCuenta@mixteco.utm.mx ";
28         String RCPT_TO = "RCPT TO: destino@nuyoo.utm.mx ";
29         String DATA = "DATA"; // inicio del mensaje
30         String ASUNTO = "Subject: Prueba Java\n";
31
32         // Nota: "\n.\n" indica el final el mensaje
33         String MENSAJE = "Cadena de mensaje\n.\n";
34
35         Socket smtp = null; // socket de SMTP
36
37         try { // Nota: 25 es el número de puerto SMTP por omisión
38             smtp = new Socket("mixteco.utm.mx", 25);
39             OutputStream os = smtp.getOutputStream();
40             ps = new PrintStream(os);
41             InputStream is = smtp.getInputStream();
```

```
42         dis= new BufferedReader(new InputStreamReader(is));
43     }
44     catch (IOException e)
45     {
46         System.out.println("Error al conectar: " + e);
47     }
48
49     try {
50         String loc = InetAddress.getLocalHost().getHostName();
51         enviar(HELO + loc);
52         recibir();           // obtener la respuesta SMTP
53         enviar(MAIL_FROM);  // enviar el remitente
54         recibir();
55         enviar(RCPT_TO);    // enviar el receptor
56         recibir();
57         enviar(DATA);       // enviar el inicio de mensaje
58         recibir();
59         enviar(ASUNTO);
60         recibir();
61         enviar(MENSAJE);
62         recibir();
63         smtp.close();
64     }
65     catch (IOException e)
66     {
67         System.out.println("Error al enviar:" + e);
68     }
69
70     System.out.println("Correo enviado!");
71 }
72 }
```

Listing 293. Ejemplo de cliente SMTP.

Un ejemplo muy claro de una aplicación cliente/servidor mediante sockets es la del juego de gato presentada por Deitel [?], donde el servidor está a la espera de que se conecten dos clientes jugadores de gato. Se muestra a continuación el lado del cliente.

Ejemplo:

```
1 // Cliente de TicTacToe con JavaFX
2 import javafx.application.Application;
```

```
3 import javafx.application.Platform;
4 import javafx.scene.Scene;
5 import javafx.scene.control.TextArea;
6 import javafx.scene.control.TextField;
7 import javafx.scene.layout.BorderPane;
8 import javafx.scene.layout.GridPane;
9 import javafx.scene.layout.StackPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Rectangle;
12 import javafx.scene.text.Font;
13 import javafx.scene.text.Text;
14 import javafx.stage.Stage;
15 import java.net.*;
16 import java.io.*;
17
18 public class TicTacToeClient extends Application implements Runnable {
19     private TextField id;
20     private TextArea display;
21     private GridPane boardPanel;
22     private Square board[][];
23     private Square currentSquare;
24     private Socket connection;
25     private DataInputStream input;
26     private DataOutputStream output;
27     private Thread outputThread;
28     private char myMark;
29     private boolean myTurn;
30
31     @Override
32     public void start(Stage primaryStage) {
33         display = new TextArea();
34         display.setEditable(false);
35         display.setPrefRowCount(4);
36         display.setPrefColumnCount(30);
37
38         boardPanel = new GridPane();
39         board = new Square[3][3];
40
41         for (int row = 0; row < 3; row++) {
42             for (int col = 0; col < 3; col++) {
43                 board[row][col] = new Square(row, col);
44                 boardPanel.add(board[row][col], col, row);
```

```
45     }
46 }
47
48 id = new TextField();
49 id.setEditable(false);
50
51 BorderPane root = new BorderPane();
52 root.setTop(id);
53 root.setCenter(boardPanel);
54 root.setBottom(display);
55
56 Scene scene = new Scene(root, 300, 300);
57 primaryStage.setTitle("Cliente Tic-Tac-Toe");
58 primaryStage.setScene(scene);
59 primaryStage.show();
60
61 // Iniciar conexión en un hilo separado
62 outputThread = new Thread(this);
63 outputThread.start();
64 }
65
66 public void run() {
67     try {
68         connection = new Socket(InetAddress.getLocalHost(), 5000);
69         input = new DataInputStream(connection.getInputStream());
70         output = new DataOutputStream(connection.getOutputStream());
71     } catch (IOException e) {
72         e.printStackTrace();
73         System.exit(1);
74     }
75
76     try {
77         myMark = input.readChar();
78         Platform.runLater(() -> id.setText("Eres el jugador \"" + myMark + "\""));
79         myTurn = (myMark == 'X');
80     } catch (IOException e) {
81         e.printStackTrace();
82     }
83
84     while (true) {
85         try {
86             String s = input.readUTF();
```

```
87         processMessage(s);
88     } catch (IOException e) {
89         e.printStackTrace();
90         break;
91     }
92 }
93 }
94
95 public void processMessage(String s) {
96     if (s.equals("Movida valida.")) {
97         Platform.runLater(() -> {
98             display.appendText("Movida valida, espere un momento.\n");
99             currentSquare.setMark(myMark);
100             currentSquare.repaint();
101         });
102     } else if (s.equals("Movida invalida, intente de nuevo")) {
103         Platform.runLater(() -> display.appendText(s + "\n"));
104         myTurn = true;
105     } else if (s.equals("El oponente movio")) {
106         try {
107             int loc = input.readInt();
108             int row = loc / 3;
109             int col = loc % 3;
110
111             Platform.runLater(() -> {
112                 board[row][col].setMark(myMark == 'X' ? 'O' : 'X');
113                 board[row][col].repaint();
114                 display.appendText("El oponente movio. Es tu turno.\n");
115                 myTurn = true;
116             });
117         } catch (IOException e) {
118             e.printStackTrace();
119         }
120     } else {
121         Platform.runLater(() -> display.appendText(s + "\n"));
122     }
123 }
124
125 public static void main(String[] args) {
126     launch(args);
127 }
128
```

```
129 // Clase interna Square
130 private class Square extends StackPane {
131     private char mark;
132     private int row, col;
133     private Text text = new Text();
134
135     public Square(int row, int col) {
136         this.row = row;
137         this.col = col;
138         this.mark = ' ';
139
140         Rectangle border = new Rectangle(50, 50);
141         border.setFill(null);
142         border.setStroke(Color.BLACK);
143
144         text.setFont(Font.font(20));
145
146         getChildren().addAll(border, text);
147
148         setOnMouseClicked(e -> {
149             if (myTurn) {
150                 currentSquare = this;
151                 try {
152                     output.writeInt(this.row * 3 + this.col);
153                     myTurn = false;
154                 } catch (IOException ex) {
155                     ex.printStackTrace();
156                 }
157             }
158         });
159     }
160
161     public void setMark(char c) {
162         mark = c;
163     }
164
165     public void repaint() {
166         text.setText(String.valueOf(mark));
167     }
168 }
169 }
```


Listing 294. Ejemplo - Cliente de TicTacToe con JavaFX.

35.2.2 Programas servidor

Veremos ahora algunos ejemplos de programas servidores, recordando que ya se ha mencionado la importancia de que estos sean capaces de soportar la conexión de varios clientes al mismo tiempo, por lo que es relevante el manejo de programación concurrente.

Inicialmente veamos el código de un servidor genérico, este no hace nada en particular, únicamente se muestra como un ejemplo de los aspectos generales de los servidores en Java.

Ejemplo:

```
1  //Ejemplo de un servidor genérico multihilado
2  import java.net.*;
3  import java.io.*;
4  import java.util.*;
5
6  public class GenericServer {
7      // 1234 puede ser cualquier número de puerto que se determine
8      int serverPort = 1234;
9      public static void main(String args[]){
10         //crear un objeto de servidor y ejecutarlo
11         GenericServer server = new GenericServer();
12         server.run();
13     }
14     public GenericServer() {
15         super();
16     }
17     public void run() {
18         try {
19             //crear un socket servidor en el puerto especificado
20             ServerSocket server = new ServerSocket(serverPort);
21             do {
22                 //hacer un ciclo infinito para aceptar conexiones entrantes
23                 Socket client = server.accept();
24                 //crear un hilo nuevo para cada conexión
25                 (new ServerThread(client)).start();
26             } while(true);
27         } catch(IOException ex) {
28             System.exit(0);
29         }
30     }
31 }
```

```
30     }
31 }
32
33 class ServerThread extends Thread {
34     Socket client;
35     //almacenar una referencia al socket en el que
36     //está conectado el cliente
37     public ServerThread(Socket client) {
38         this.client = client;
39     }
40     //este es el método inicial del hilo
41     public void run() {
42         try {
43             //crea flujos para comunicarse con el cliente
44             ServiceOutputStream outStream = new ServiceOutputStream(
45                 new BufferedOutputStream(client.getOutputStream()));
46             ServiceInputStream inStream = new ServiceInputStream(client.getInputStream());
47             //leer la solicitud del cliente en el flujo de entrada
48             ServiceRequest request = inStream.getRequest();
49             //procesar solicitudes del cliente y devolver la salida al cliente
50             while (processRequest(outStream)) {};
51         } catch (IOException ex) {
52             System.exit(0);
53         }
54         try {
55             client.close();
56         } catch (IOException ex) {
57             System.exit(0);
58         }
59     }
60     //procesamiento de solicitudes
61     public boolean processRequest(ServiceOutputStream outStream) {
62         return false;
63     }
64 }
65
66 //filtro de flujo de entrada
67 class ServiceInputStream extends FilterInputStream {
68     public ServiceInputStream(InputStream in) {
69         super(in);
70     }
71     //método para leer solicitudes de los clientes en el flujo de entrada
```

```
72     public ServiceRequest getRequest() throws IOException {
73         ServiceRequest request = new ServiceRequest();
74         return request;
75     }
76 }
77
78 //filtro de flujo de salida
79 class ServiceOutputStream extends FilterOutputStream {
80     public ServiceOutputStream(OutputStream out) {
81         super(out);
82     }
83 }
84
85 //clase que implementa solicitudes del cliente
86 class ServiceRequest {
87 }
```

Listing 295. Ejemplo de un servidor genérico multihilado.

El programa servidor del gato, el cual lógicamente debe ejecutarse antes que los clientes para estar listo a recibir las conexiones. Es importante señalar que estos ejemplos no contienen más que una validación simple de las casillas ocupadas y el turno. No realiza por ejemplo, una validación para ver si alguno de los jugadores gana el juego.

Ejemplo:

```
1  // Servidor de TicTacToe.
2  import java.awt.*;
3  import java.net.*;
4  import java.io.*;
5
6  public class TicTacToeServer extends Frame {
7      private byte board[];
8      private boolean xMove;
9      private TextArea output;
10     private Player players[];
11     private ServerSocket server;
12     private int numberOfPlayers;
13     private int currentPlayer;
14
15     public TicTacToeServer ()
16     {
```

```
17     super( "Servidor Tic-Tac-Toe" );
18     board = new byte[ 9 ];
19     xMove = true;
20     players = new Player[ 2 ];
21     currentPlayer = 0;
22
23     try {
24         server = new ServerSocket( 5000, 2 );
25     }
26     catch( IOException e ) {
27         e.printStackTrace();
28         System.exit( 1 );
29     }
30
31     output = new TextArea();
32     add( "Center", output );
33     resize( 300, 300 );
34     show();
35 }
36
37 // espera por dos conexiones de los clientes
38 public void execute()
39 {
40     for ( int i = 0; i < players.length; i++ ) {
41         try {
42             players[ i ] =
43                 new Player( server.accept(), this, i );
44             players[ i ].start();
45             ++numberOfPlayers;
46         }
47         catch( IOException e ) {
48             e.printStackTrace();
49             System.exit( 1 );
50         }
51     }
52 }
53
54 public int getNumberOfPlayers() {
55     return numberOfPlayers;
56 }
57
58 public void display( String s )
```

```
59     {
60         output.appendText( s + "\n" );
61     }
62
63     public synchronized boolean validMove( int loc, int player )
64     {
65         boolean moveDone = false;
66
67         while ( player != currentPlayer ) {
68             try {
69                 wait();
70             }
71             catch( InterruptedException e ) {
72             }
73         }
74
75         if ( !isOccupied( loc ) ) {
76             board[ loc ] =
77                 (byte)( currentPlayer == 0 ? 'X' : 'O' );
78             currentPlayer = ++currentPlayer % 2;
79             players[ currentPlayer ].otherPlayerMoved( loc );
80             notify();    // indicar al jugador en espera que continúe
81             return true;
82         }
83         else
84             return false;
85     }
86
87     public boolean isOccupied( int loc )
88     {
89         if ( board[ loc ] == 'X' || board [ loc ] == 'O' )
90             return true;
91         else
92             return false;
93     }
94
95     public boolean gameOver()
96     {
97         return false;
98     }
99
100    public boolean handleEvent( Event event )
```

```
101     {
102         if ( event.id == Event.WINDOW_DESTROY ) {
103             hide();
104             dispose();
105
106             System.exit( 0 );
107         }
108
109         return super.handleEvent( event );
110     }
111
112     public static void main( String args[] )
113     {
114         TicTacToeServer game = new TicTacToeServer();
115
116         game.execute();
117     }
118 }
119
120 class Player extends Thread {
121     Socket connection;
122     DataInputStream input;
123     DataOutputStream output;
124     TicTacToeServer control;
125     int number;
126     char mark;
127
128     public Player( Socket s, TicTacToeServer t, int num )
129     {
130         mark = ( num == 0 ? 'X' : 'O' );
131
132         connection = s;
133
134         try {
135             input = new DataInputStream(
136                 connection.getInputStream() );
137             output = new DataOutputStream(
138                 connection.getOutputStream() );
139         }
140         catch( IOException e ) {
141             e.printStackTrace();
142             System.exit( 1 );
```

```

143     }
144
145     control = t;
146     number = num;
147 }
148
149 public void otherPlayerMoved( int loc )
150 {
151     try {
152         output.writeUTF( "El oponente movio" );
153         output.writeInt( loc );
154     }
155     catch ( IOException e ) {}
156 }
157
158 public void run()
159 {
160     boolean done = false;
161
162     try {
163         control.display( "Jugador " +
164             ( number == 0 ? 'X' : 'O' ) + " conectado" );
165         output.writeChar( mark );
166         output.writeUTF( "Jugador " +
167             ( number == 0 ? "X conectado\n" :
168                 "O conectado, espere un momento\n" ) );
169
170         if ( control.getNumberOfPlayers() < 2 ) {
171             output.writeUTF( "Esperando otro jugador" );
172
173             while ( control.getNumberOfPlayers() < 2 )
174                 ;
175
176             output.writeUTF(
177                 "Ya se conectó otro jugador. Es tu turno." );
178         }
179
180         while ( !done ) {
181             int location = input.readInt();
182
183             if ( control.validMove( location, number ) ) {
184                 control.display( "pos: " + location );

```

```
185         output.writeUTF( "Movida válida." );
186     }
187     else
188         output.writeUTF( "Movida inválida, intente de nuevo" );
189
190     if ( control.gameOver() )
191         done = true;
192 }
193 connection.close();
194 }
195 catch( IOException e ) {
196     e.printStackTrace();
197     System.exit( 1 );
198 }
199 }
200 }
```

Listing 296. Ejemplo - Servidor de TicTacToe.

Referencias

- [1] Weitzenfeld A (1994) Paradigma orientado a objetos. *División Académica de Computación Depto Académico de Computación ITAM México* .
- [2] Müller P (1997) Introduccion a la programacion orientada a objetos empleando c++. *Globewide Network Academy (GNA)* .
- [3] Sun Microsystems (1997) The java language: An overview (Sun Microsystems - Java White Paper),
- [4] Wegner P (1986) Classification in object-oriented systems. *ACM Sigplan Notices* (ACM), Vol. 21, pp 173–182.
- [5] Deitel HM, Deitel PJ, Deitel A (2012) *C How to Program: International Edition* (Pearson Education UK), .
- [6] Lutz M (2013) *Learning python: Powerful object-oriented programming* (.o'Reilly Media, Inc."), .
- [7] Timothy B (2008) *An Introduction to object-oriented programming* (Pearson Education India), .
- [8] FAR (1996) *Programacion Orientada a Objetos en C+* (Fundación Arturo Rosenblueth), .
- [9] Pilgrim M, Willison S (2009) *Dive into python 3*. Vol. 2 (Springer), .
- [10] Meyer B (1997) *Object-oriented software construction*. Vol. 2 (Prentice hall Englewood Cliffs), .
- [11] McAffer J, Lemieux JM, Aniszczyk C (2010) *Eclipse rich client platform* (Addison-Wesley Professional), .
- [12] Deitel HM, Deitel PJ (2006) *C++ how to program* (Prentice Hall), .
- [13] Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen WE, et al. (1991) *Object-oriented modeling and design*. Vol. 199 (Prentice-hall Englewood Cliffs, NJ), .
- [14] Strachey C (2000) Fundamental concepts in programming languages. *Higher-order and symbolic computation* 13(1-2):11–49.
- [15] Cardelli L, Wegner P (1985) On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17(4):471–523.
- [16] Josuttis NM, Gregor D (2003) *C++ Templates: The Complete Guide*. Vol. 338 (Addison-Wesley Professional Boston), .
- [17] Graphics S (1997) Standard template library programmer's guide.

- [18] Bracha G (2004) Generics in the java programming language.
- [19] Kreft K, Langer A (2004) Language features of java generics (Fawcette Technical Publications), Available at http://www.ftponline.com/javapro/2004_03/online/jgen_kkreft_03_03_04/default_pf.aspx.
- [20] Turner K (2001) Catching more errors at compile time with generic java. Accessed: 2007-06-01 Available at <http://www.ibm.com/developerworks/library/j-genjava.html>.
- [21] Naftalin M, Wadler P (2007) *Java generics and collections* (.O'Reilly Media, Inc.), .
- [22] Gosling J, Holmes DC, Arnold K (2005) *The Java programming language* (Addison-Wesley), .
- [23] Microsystems S (2002) Programming with assertions. Accessed: 2007-06.01 Available at <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>.
- [24] Turing AM (1937) Computability and λ -definability. *The Journal of Symbolic Logic* 2(4):153–163.
- [25] Liang YD (2015) *Introduction to Java programmng: comprehensive version* (Pearson Education), 10th Ed.
- [26] MATÍAS IAC (2003) *HERRAMIENTA CASE PARA LA GENERACIÓN DE CÓDIGO C++ A PARTIR DE DIAGRAMAS DE CLASE UML*. Ph.D. thesis. UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA, .
- [27] Fernández y Fernández CA, et al. (2002) Modelado visual con uml. *REPOSITORIO NACIONAL CONACYT* .

Apéndice A. Herramientas adicionales sugeridas

BlueJ

BlueJ³ es un programa desarrollado por la universidades de *Kent* y *Deakin* para ayudar a los estudiantes a entender programación orientada a objetos en Java, particularmente ayuda a entender la herencia.

A partir de un diagrama de clases, BlueJ puede generar el código básico de la clase en Java, el cuál puede ser editado y compilado conforme las necesidades del programa. El programa es básico -ver figura ?? - y fácil de usar permitiendo entender estructuras complejas en las relaciones de herencia.

El código Java generado por BlueJ para el diagrama de la figura anterior es el siguiente:

```
1  /**
2   * Write a description of class Vehiculo here.
3   *
4   * @author (your name)
5   * @version (a version number or a date)
6   */
7  public class Vehiculo
8  {
9      // instance variables - replace the example below with your own
10     private int x;
11
12     /**
13      * Constructor for objects of class Vehiculo
14      */
15     public Vehiculo()
16     {
```

³Ver: <http://www.bluej.org/>

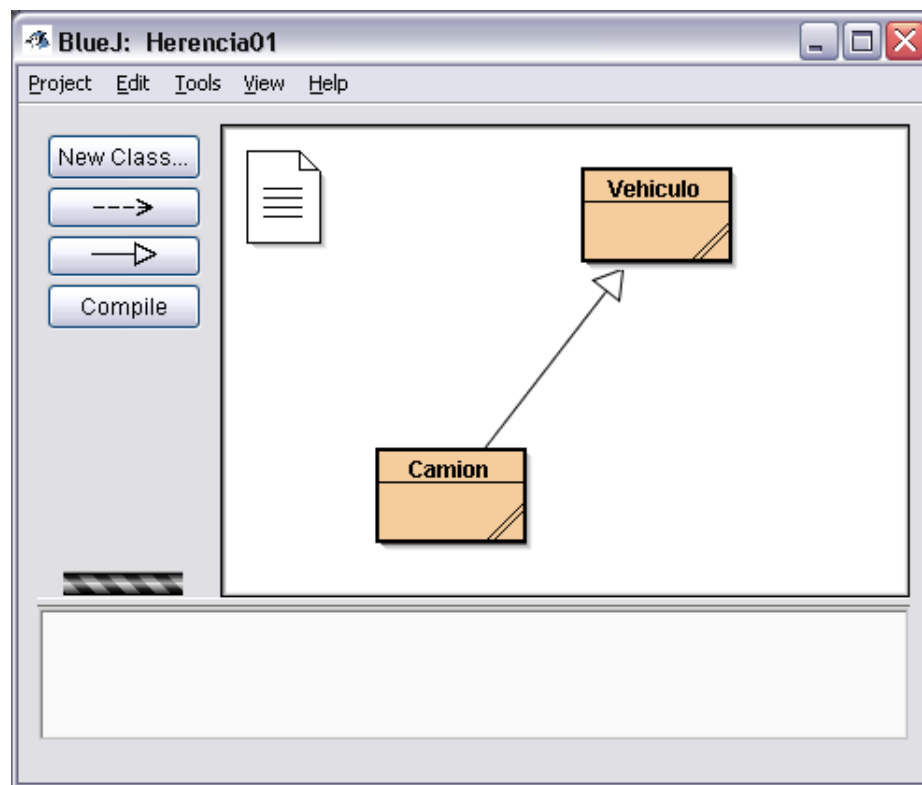


Fig. 35.1. Interface de BlueJ

```
17         // initialise instance variables
18         x = 0;
19     }
20
21     /**
22     * An example of a method - replace this comment with your own
23     *
24     * @param y    a sample parameter for a method
25     * @return     the sum of x and y
26     */
27     public int sampleMethod(int y)
28     {
29         // put your code here
30         return x + y;
31     }
32 }
33
34
35
36 /**
37  * Write a description of class Camion here.
38  *
39  * @author (your name)
40  * @version (a version number or a date)
41  */
42 public class Camion extends Vehiculo
43 {
44     // instance variables - replace the example below with your own
45     private int x;
46
47     /**
48     * Constructor for objects of class Camion
49     */
50     public Camion()
51     {
52         // initialise instance variables
53         x = 0;
54     }
55
56     /**
57     * An example of a method - replace this comment with your own
58     *
```

```
59      * @param y    a sample parameter for a method
60      * @return     the sum of x and y
61      */
62      public int sampleMethod(int y)
63      {
64          // put your code here
65          return x + y;
66      }
67  }
```

Listing 297. Ejemplo de código generado por BlueJ.

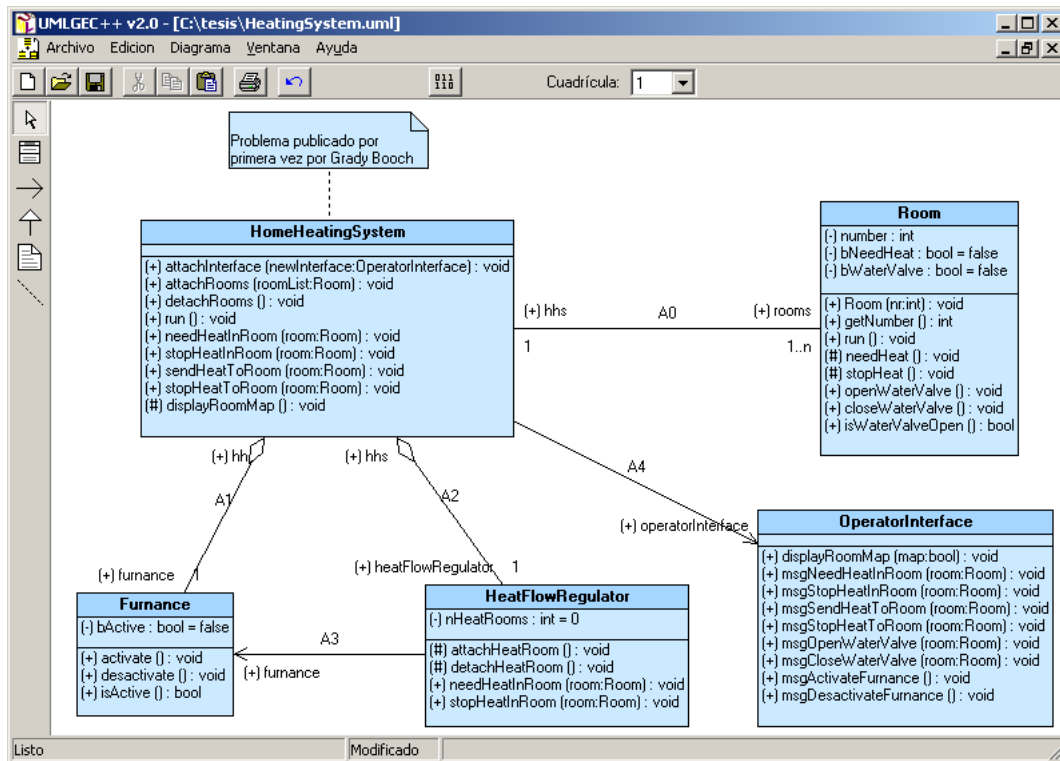


Fig. 35.2. Herramienta CASE UMLGEC++

UMLGEC++

El proyecto de desarrollo de esta herramienta CASE (UMLGEC ++)[? ?] soporta la notación UML⁴ para diagramas de clase y generación de código en C++, con una interfaz lo más completa y sencilla posible, ver figura ?? . Siendo útil para entender gráficamente conceptos básicos de objetos y su correspondiente implementación en código. Los elementos de este software son:

- Depósito de datos
- Módulo para Creación de Diagramas y Modelado
- Generador de código
- Analizador de sintaxis

De la generación de código se puede decir:

- A partir del diagrama se genera la estructura de las clases.

⁴Información básica sobre UML puede ser vista en [?]

- Se crean automáticamente: el constructor, el constructor de copia, el operador de asignación, las operaciones de igualdad y el destructor.
- Todos los atributos y asociaciones son establecidos como privados independientemente de la visibilidad establecida por el usuario, pero el acceso a ellos está permitido mediante operaciones *get* y *set* generadas automáticamente para cada atributo o asociación, las cuáles adquieren la visibilidad correspondiente al atributo o asociación al que hacen referencia.
- Se definen los cuerpos de las operaciones *get* y *set*, como funciones *inline*.