POO Handbook 2025

Notas de Programación Orientada a Objetos con

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, Infierno, Canto XXVI

Universidad Tecnológica de la Mixteca 2025
POO Handbook 2025, ?? páginas (26 de noviembre de 2025)

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.

Listing 1.1. Groovy: Hola Mundo

```
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.

Listing 1.2. JRuby: Hola Mundo

```
puts "Hola Mundo"
```

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

Listing 1.3. Jython: Hola Mundo

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

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

Listing 1.4. Kotlin: Hola Mundo

```
fun main() {
    println("Hola Mundo")
}
```

- **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#.

Listing 1.5. IronPython: Hola Mundo

```
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.

Listing 1.6. Dart: Hola Mundo

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

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

Listing 1.7. Go: Hola Mundo

```
package main
import "fmt"
func main() {
    fmt.Println("Hola Mundo")
}
```

- **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.

Listing 1.8. Swift: Hola Mundo

```
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.

Listing 1.9. Cobra: Hola Mundo

```
class Hola
    def main
        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/

Listing 1.10. Fantom: Hola Mundo

```
class Hola {
  static Void main() {
    echo("Hola Mundo")
  }
}
```

- **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.

Listing 1.11. Elixir: Hola Mundo

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

En la tabla **??** podemos ver una tabla con algunas características de los languajes 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.

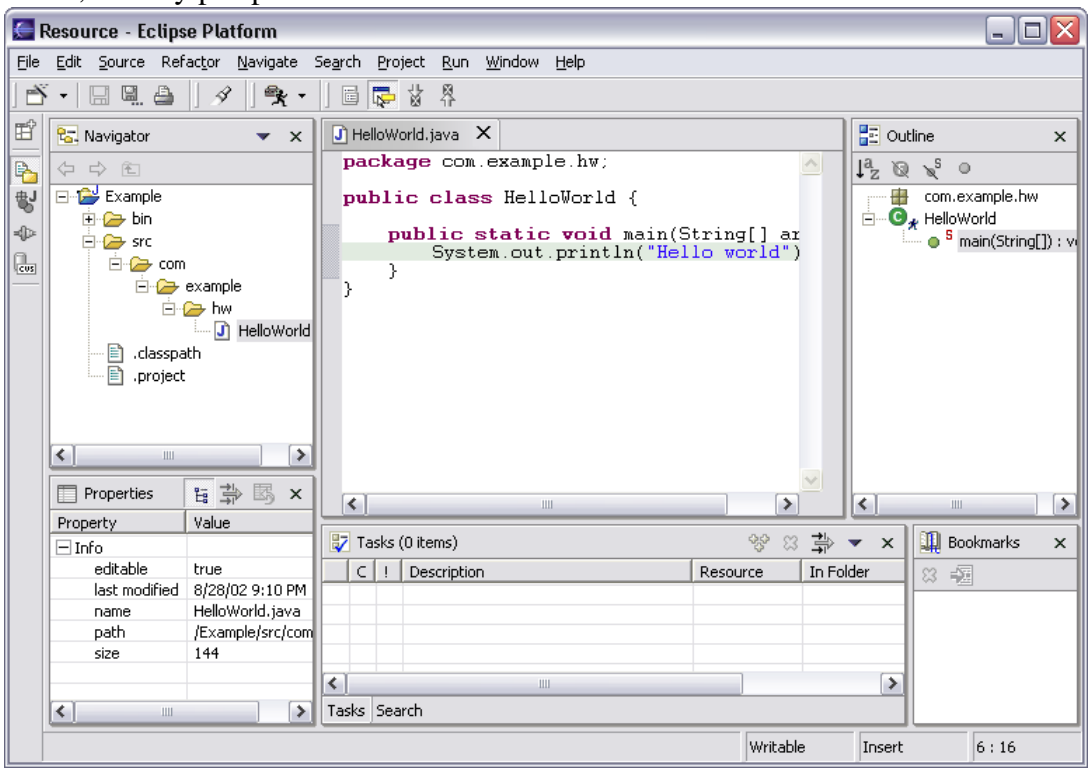
Eclipse

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 "*workbenchs*" 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

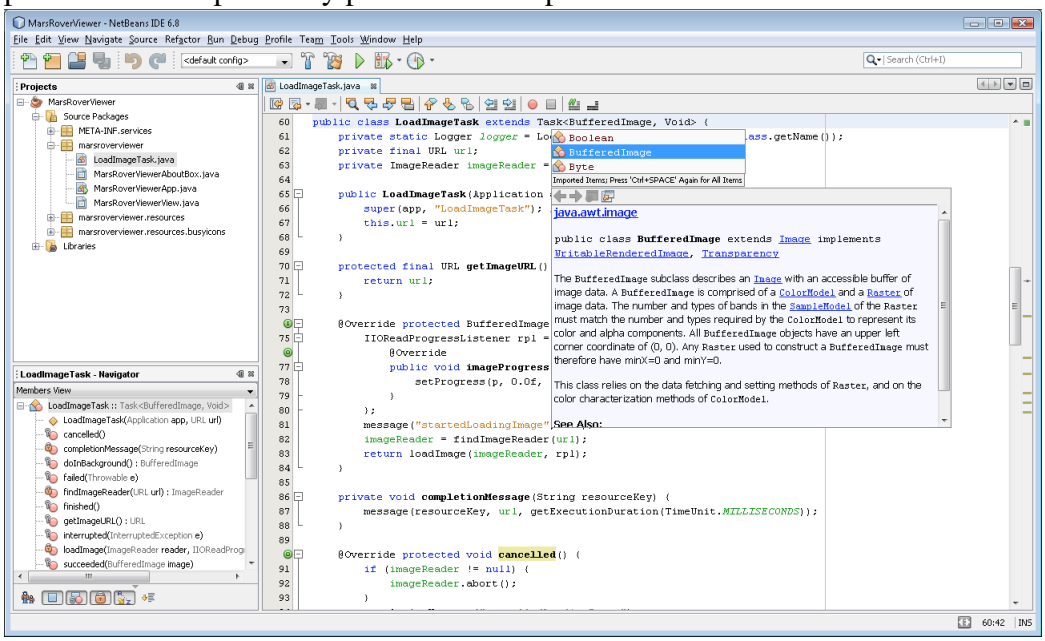
Mono

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.

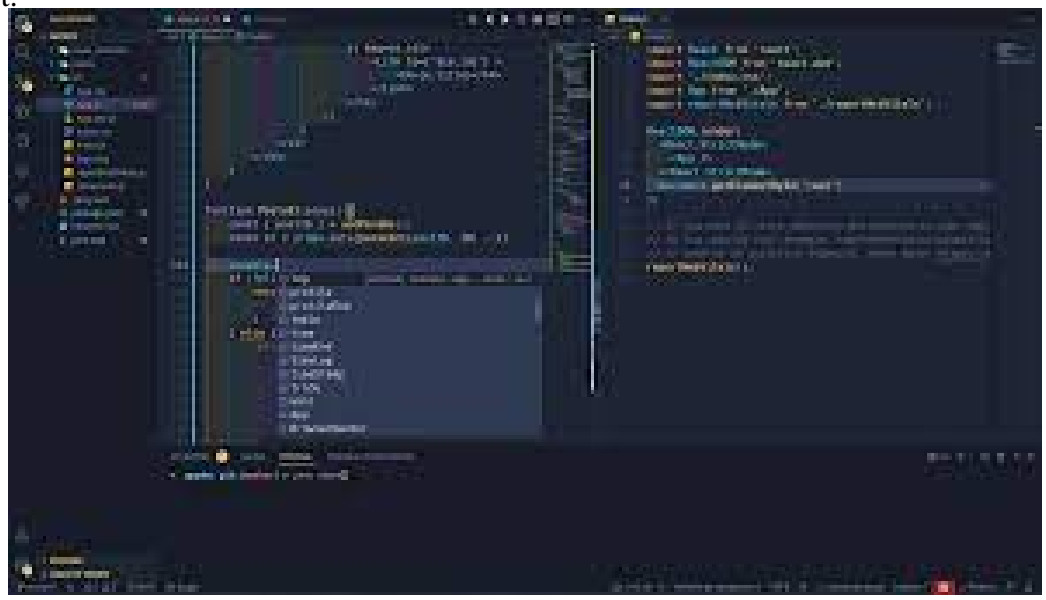
Creado en 1996 como un proyecto universitario en la Universidad Karlova (Praga), fue adoptado más tarde por Sun y posteriormente por Oracle.



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]



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.

Algunos editores ligeros

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

- Geany. [2]

[1] https://code.visualstudio.com/
[2] https://www.geany.org/

- 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.
- **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.

[3] https://www.sublimetext.com/
[4] https://atom.io/
[5] http://brackets.io/

Comparación de IDEs

En la tabla **??**, 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.

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 da 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 salida estándar. Que se apoya del operador $\ll$, el cual se conoce como operador de inserción de flujo *colocar en*.
- *cin*. Utiliza el flujo de entrada estándar. Que se apoya del operador $\gg$, 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 `inline <declaración de la función>`

Ejemplo:

```
inline float suma (float a, float b) {
       return a+b;
}

inline int max( int a, int b) {
                return (a > b) ? a :  b;
}
```

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:

```
#include <iostream>

int main() {
        int i=0;

        for (i=1; i<10; i++){
                int j=10;
                std::cout<<i<<" j: "<<j<<std::endl;
           }

           std::cout<<"\ni al salir del ciclo: "<<i;

        return 0;
}
```

Listing 1. Ejemplo declaración de variables

O usando el área de nombres estándar:

```
#include <iostream>

using namespace std;

int main() {
        int i=0;

        for (i=1; i<10; i++){
                int j=10;
                cout<<i<<" j: "<<j<<endl;
           }

           cout<<"\ni al salir del ciclo: "<<i;

        return 0;
}
```

Listing 2. Ejemplo declaración de variables 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.

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:

```
float h;

void g(int h) {
        float a;
        int b;

        a=::h;        // a se inicializa con la variable global h

```

```
9           b=h;        // b se inicializa con la variable local h
10  }
```

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:

```
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 3. Ejemplo de valores por default

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;
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 4. Valores o argumentos por deafult.

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:

```
// Comparando parámetros por valor, por valor con  apuntadores ("referencia"),
// y paso por referencia real

#include <iostream>
using namespace std;

int porValor(int);
void porApuntador(int *);
void porReferencia( int &);

int main() {
        int x=2;
        cout << "x= " << x << " antes de llamada a porValor \n"
                << "Regresado por la función: "<< porValor(x)<<endl
                << "x= " << x << " despues de la llamada a porValor\n\n";

        int y=3;
        cout << "y= " << y << " antes de llamada a porApuntador\n";
        porApuntador(&y);
        cout << "y= " << y << " despues de la llamada a porApuntador\n\n";

        int z=4;
        cout << "z= " << z << " antes de llamada a porReferencia \n";
        porReferencia(z);
        cout<< "z= " << z << " despues de la llamada a porReferencia\n\n";
        return 0;
}

int porValor(int valor){
        return valor*=valor;     //parámetro no modificado
}

void porApuntador(int *p){
        *p *= *p;                // parámetro modificado
}

void porReferencia( int &r){
        r *= r;                 //parámetro modificado
}

```

Listing 5. 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:

```
int max=1000, &sMax=max;        //declaro max y sMax es un alias de max
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:

```
// variable por referencia
#include <iostream>

using namespace std;

int main() {
        int x=2, &y=x, z=8;

        cout << "x= "<<x <<endl
                <<"y= "<<y<<endl;

        y=10;
        cout << "x= "<<x <<endl
                <<"y= "<<y<<endl;
// Reasignar no esta permitido
//          \&  y= &z;
//        cout << "z= "<<z <<endl
//                <<"y= "<<y<<endl;

        y++;
        cout << "z= "<<z <<endl
                <<"y= "<<y<<endl;

        return 0;
}
```

Listing 6. 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 (*NULL* 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 7. 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 8. 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 9. 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 hacia 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 10. 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 11. 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 asignaci¢n lo corta en entero.
30         cout<<c<<endl;
31
32         return 0;
33 }
```

Listing 12. 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:

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 13. Ejemplo de enumeración en C++.

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

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 un alcance para identificadores tales como tipos, funciones, variables, etc. Algunas características[2]:

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

[2]Más información: C++ reference: namespace

```
16      cout << ns2::x << '\n';
17  }
18
```

Listing 14. 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 compiladortes 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 15. 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 16. 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.

2.15. Características relevantes de C++17 [3]

C++17 es la séptima versión del estándar de C++, fue aprobada en diciembre de 2017, y introduce varias características y mejoras en el lenguaje, algunas de las más importantes incluyen:

- **Inicialización de estructuras y clases**: Se permite inicializar estructuras y clases de manera similar a como se inicializan los arreglos y los objetos de cualquier tipo de contenedor. Esto se logra mediante la inclusión de una lista de inicializadores entre llaves {}.
- **Inicialización automática de variables**: Se permite inicializar variables automáticamente en el momento de su declaración, ahorrando la necesidad de escribir código adicional para inicializarlas.
- **Eliminación de if y switch**: Se permite eliminar el cuerpo de un if o un switch si su condición es siempre verdadera o falsa, respectivamente.
- **Programación concurrente y paralela**: Se ha añadido soporte para la programación concurrente y paralela mediante el uso de las librerías estándar de C++.
- **Mejoras en las plantillas**: Se han añadido mejoras en las plantillas, como la posibilidad de especificar un tipo de retorno automático para las funciones plantilla, lo que permite una mayor flexibilidad y legibilidad en el código.
- **Mejoras en la sintaxis**: Se han añadido mejoras en la sintaxis del lenguaje, como la posibilidad de usar el operador .ªuto"para deducir el tipo de una variable en una expresión.

C++17 ha mejorado significativamente el lenguaje, haciéndolo más conciso, fácil de leer y de escribir, y más adecuado para la programación concurrente y paralela. Estas mejoras hacen que el código sea más eficiente y escalable, y que sea más fácil para los desarrolladores escribir código correcto y seguro.

[3] Partes de esta sección fueron desarrollados con apoyo de ChatGPT

2.15.1. Inicialización de estructuras y clases

La inicialización de estructuras y clases es una característica introducida en C++17 que permite inicializar los miembros de una estructura o clase mediante una lista de inicializadores entre llaves {}. Esto es similar a cómo se inicializan los arreglos y los objetos de cualquier tipo de contenedor.

La sintaxis para inicializar una estructura es la siguiente:

```
1
2 struct MyStruct {
3     int x;
4     float y;
5 };
6
7 MyStruct myStruct {1, 3.14f};
8
```

Para una clase:

```
1
2 class MyClass {
3 public:
4     int x;
5     float y;
6 };
7
8 MyClass myClass {1, 3.14f};
9
```

En el ejemplo anterior se define una estructura llamada *MyStruct* con dos miembros: ”x” e ”y”, se declara una variable *myStruct* de ese tipo y se inicializa con los valores 1 y 3.14f respectivamente. Y también se define una clase llamada *MyClass* con dos miembros: ”x” e ”y”, se declara una variable *myClass* de ese tipo y se inicializa con los valores 1 y 3.14f respectivamente.

Además, es posible inicializar solo algunos de los miembros de una estructura o una clase, dejando los demás con su valor por defecto:

```
1 MyStruct myStruct {1};  // x = 1, y = 0.0f (valor por defecto)
2
3 MyClass myClass {1};   // x = 1, y = 0.0f (valor por defecto)
4
```

La inicialización de estructuras y clases es una característica muy útil que permite escribir código más conciso y legible, y también ayuda a prevenir errores comunes relacionados con la inicialización de variables.

2.15.2. Inicialización automática de variables

La inicialización automática de variables es una característica introducida en C++17 que permite inicializar una variable automáticamente en el momento de su declaración, sin necesidad de escribir código adicional para inicializarla. Esto se logra mediante el uso del operador *auto* junto con el valor con el que se desea inicializar la variable.

La sintaxis para inicializar una variable automáticamente es la siguiente:

Sintaxis
`auto <variable>= <valor>;`

Se utiliza el operador *auto* para inicializar automáticamente la variable con el valor, y el compilador deduce automáticamente el tipo de la variable.

Se puede usar auto con cualquier tipo de dato, incluso con tipos compuestos como *structs* o clases:

```
struct MyStruct {
    int x;
    float y;
};

auto myStruct = MyStruct{1, 3.14f};

class MyClass {
public:
    int x;
    float y;
};

auto myClass = MyClass{1, 3.14f};
```

En estos ejemplos, se utiliza el operador *auto* para inicializar automáticamente las variables *myStruct* y *myClass* con los valores especificados en las estructuras *MyStruct* y *MyClass* respectivamente.

La inicialización automática de variables es una característica muy útil que permite escribir código más conciso y legible, ya que el compilador deduce automáticamente el tipo de la variable a partir del valor con el que se inicializa. Esto también ayuda a prevenir errores comunes relacionados con la inicialización de variables, ya que el compilador se encarga de realizar la deducción del tipo de forma segura.

2.15.3. Eliminación de sentencias vacias if y switch

En C++17 se ha introducido la característica de ”Eliminación de sentencias vacías” en las estructuras de control *if* y *switch*.

Se refiere a la eliminación de las sentencias vacías de las estructuras de control *if* y *switch* cuando su condición es siempre verdadera o falsa.

Por ejemplo, consideremos el siguiente código:

```
1  int x = 5;
2
3  if (x > 0)
4      x = x + 2;
5  else
6      ; // sentencia vacía
7
```

En este caso, la condición del *if* es siempre verdadera, por lo que el cuerpo del *else* no se ejecutará nunca. Es decir, la sentencia vacía ";" en el cuerpo del *else* no tiene efecto alguno. El compilador puede eliminarla automáticamente y generar código más eficiente.

Con respecto al *switch*, consideremos el siguiente código:

```
1  int x = 5;
2
3  switch (x) {
4      case 5:
5          x = x + 2;
6          break;
7      case 6:
8          ; // sentencia vacía
9          break;
10     default:
11         x = x - 2;
12 }
```

En este caso, el valor de x es siempre 5, por lo que el cuerpo de case 6 no se ejecutará nunca, es decir la sentencia vacía ";"no tiene efecto alguno. El compilador puede eliminarla automáticamente y generar código más eficiente.

La eliminación de sentencias vacías ayuda a escribir código más conciso y legible ya que permite eliminar las estructuras de control innecesarias. Además, también ayuda a mejorar el rendimiento del código ya que el compilador puede generar código más eficiente al eliminar las sentencias vacías.

2.15.4. Mejoras en las plantillas

Las plantillas son una característica de C++ que permite escribir código genérico que se aplica a diferentes tipos de datos. C++17 ha introducido varias mejoras en las plantillas, que hacen que sea más fácil y flexible escribir código genérico.

Una de las mejoras más importantes es la posibilidad de especificar un tipo de retorno automático para las funciones plantilla. Esto permite que el compilador deduzca automáticamente el tipo de retorno de una función plantilla a partir de los argumentos de entrada.

La sintaxis para especificar un tipo de retorno automático para una función plantilla es la siguiente:

```
template <typename T>
auto add(T a, T b) -> decltype(a + b) {
    return a + b;
}

```

En este ejemplo, la función *add* es una plantilla que acepta dos argumentos de cualquier tipo ”T” y devuelve el resultado de la suma de estos argumentos. La función utiliza el operador *decltype* para especificar que el tipo de retorno de la función es el mismo que el tipo de los argumentos de entrada.

Otra mejora importante en las plantillas es la posibilidad de especificar parámetros de plantilla de forma no estricta. Esto significa que se pueden especificar parámetros de plantilla que no son necesariamente tipos.

La sintaxis para especificar parámetros de plantilla no estrictos es la siguiente:

```
template <typename T, int N>
void print_array(T (&arr)[N]) {
    for (int i = 0; i < N; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

En este ejemplo, la función *print_array* es una plantilla que acepta un arreglo *arr* de cualquier tipo ”T” y un entero ”N” que indica el tamaño del arreglo. La función imprime el contenido del arreglo en pantalla.

Estas mejoras en las plantillas hacen que el código sea más fácil y flexible de escribir, ya que permite deducir automáticamente el tipo de retorno y no solo utilizar tipos como parámetros. Esto permite una mayor flexibilidad y legibilidad en el código.

2.15.5. Mejoras en la sintaxis

C++17 ha introducido varias mejoras en la sintaxis del lenguaje que hacen que sea más fácil y legible escribir código. Algunas de las mejoras más importantes son:

Inicializadores de lista en la declaración de variables: Es posible inicializar una variable en el momento de su declaración utilizando una lista de inicializadores entre llaves . Esto es similar a cómo se inicializan los arreglos y los objetos de cualquier tipo de contenedor.

```
int x = {5};
std::vector<int> v = {1, 2, 3};
```

Mejoras en el operador ternario: El operador ternario se ha mejorado para permitir la asignación y el llamado a funciones dentro de su sintaxis.

```
int x = 5;
int y = x > 0 ? x + 2 : x - 2;

std::string message = x > 0 ? "Positive" : "Negative";

```

Mejoras en el operador de propagación: El operador de propagación "..."permite pasar los elementos de un contenedor como argumentos individuales a una función o constructor.

```
std::vector<int> v = {1, 2, 3};
std::initializer_list<int> l = {4, 5, 6};

foo(1, 2, 3);
foo(v.begin(), v.end());
foo(l);

```

Mejoras en el uso de corchetes y llaves: La sintaxis de corchetes y llaves se ha mejorado para permitir su uso como una forma de expresar bloques de código y para inicializar variables.

```
for (int i : {1, 2, 3}) {
    std::cout << i << std::endl;
}

```

Estas mejoras en la sintaxis ayudan a escribir código más legible y fácil de entender, y también ayudan a prevenir errores comunes.

2.15.6. Manejo de cadenas con *string*

Las cadenas de caracteres son elementos fundamentales en la programación, y en C++, se gestionan a través de la clase *std::string*. Esta clase proporciona una interfaz conveniente para manipular cadenas de manera eficiente. A continuación, se presentan un ejemplo completo que ilustran cómo utilizar cadenas en C++.

```
1
2  #include <iostream>
3  #include <string>
4
5  int main() {
6      // Crear una cadena e inicializarla
7      std::string cadena = "Hola, mundo";
8
9      // Imprimir la cadena original
10     std::cout << "Cadena original: " << cadena << std::endl;
11
12     // Obtener la longitud de la cadena
13     size_t longitud = cadena.length();
14     std::cout << "Longitud de la cadena: " << longitud << " caracteres" << std::endl;
15
16     // Acceder a caracteres individuales mediante índices
17     char primerCaracter = cadena[0];
18     char ultimoCaracter = cadena[longitud - 1];
19     std::cout << "Primer caracter: " << primerCaracter << std::endl;
20     std::cout << "Último caracter: " << ultimoCaracter << std::endl;
21
22     // Concatenar cadenas
23     std::string otraCadena = "¡Bienvenido!";
24     std::string concatenacion = cadena + " " + otraCadena;
25     std::cout << "Cadenas concatenadas: " << concatenacion << std::endl;
26
27     // Comparar cadenas
28     std::string otraCadena2 = "Hola, mundo";
29     if (cadena == otraCadena2) {
30         std::cout << "Las cadenas son iguales" << std::endl;
31     } else {
32         std::cout << "Las cadenas son diferentes" << std::endl;
33     }
34
35     std::string cadena1 = "Hola";
36     std::string cadena2 = "Hola";
37     int comparacion = cadena1.compare(cadena2); // comparacion = 0
38     std::cout << "Comparación: "<<comparacion << std::endl;
39
40     // Encontrar subcadenas
41     size_t posicion = cadena.find("mundo");
```

```
42      if (posicion != std::string::npos) {
43          std::cout << "La subcadena 'mundo' comienza en la posición: "
44            << posicion << std::endl;
45      } else {
46          std::cout << "La subcadena 'mundo' no se encontró" << std::endl;
47      }
48
49      // Se pueden convertir cadenas a otros tipos de datos
50      // utilizando métodos como stoi o stof
51      std::string numero = "123";
52      int valor = std::stoi(numero); // valor = 123
53      std::cout << "Valor: " << valor << std::endl;
54
55      std::string decimal = "3.1415";
56      float pi = std::stof(decimal); // pi = 3.1415
57      std::cout << "pi: " << pi << std::endl;
58
59      // Conversión a cadena con el método to_string()
60      cadena = std::to_string(pi);
61      std::cout << cadena << std::endl;
62
63      return 0;
64  }
65
```

Listing 17. Ejemplo de uso de *string* en C++.

En este programa, se ha creado una cadena inicial, se han explorado diversas operaciones comunes de cadenas y se han comentado para proporcionar claridad. Desde la creación e inicialización de cadenas hasta la comparación y búsqueda de subcadenas, estos ejemplos abarcan aspectos esenciales del uso de cadenas en C++. *size_t* es un tipo de datos sin signo en C++ utilizado para representar el tamaño o la longitud de estructuras de datos, como arreglos o cadenas.

Capítulo 3

Introducción a características de C++

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

3.1. 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.

3.2. 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 da 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 salida estándar. Que se apoya del operador $\ll$, el cual se conoce como operador de inserción de flujo *colocar en*.
- *cin*. Utiliza el flujo de entrada estándar. Que se apoya del operador $\gg$, 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';
```

3.3. 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 `inline <declaración de la función>`

Ejemplo:

```
inline float suma (float a, float b) {
       return a+b;
}

inline int max( int a, int b) {
                return (a > b) ? a :  b;
}
```

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.

3.4. 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:

```
#include <iostream>

int main() {
        int i=0;

        for (i=1; i<10; i++){
                int j=10;
                std::cout<<i<<" j: "<<j<<std::endl;
           }

           std::cout<<"\ni al salir del ciclo: "<<i;

        return 0;
}
```

Listing 18. Ejemplo declaración de variables

O usando el área de nombres estándar:

```
#include <iostream>

using namespace std;

int main() {
        int i=0;

        for (i=1; i<10; i++){
                int j=10;
                cout<<i<<" j: "<<j<<endl;
           }

           cout<<"\ni al salir del ciclo: "<<i;

        return 0;
}
```

Listing 19. Ejemplo declaración de variables 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.

3.5. 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:

```
float h;

void g(int h) {
        float a;
        int b;

        a=::h;        // a se inicializa con la variable global h

```

```
9           b=h;        // b se inicializa con la variable local h
10  }
```

3.6. 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:

```
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 20. Ejemplo de valores por default

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;
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 21. Valores o argumentos por deafult.

3.7. 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 22. 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 23. Ejemplo de variables por referencia.

3.8. 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 (*NULL* 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 24. 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 25. 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 26. Ejemplo 3 de asignación de memoria en C++

3.9. 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 hacia 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 27. 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 28. 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 asignaci¢n lo corta en entero.
30         cout<<c<<endl;
31
32         return 0;
33 }
```

Listing 29. Ejemplo 3 de uso de plantillas en C++.

3.10. 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:

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 30. Ejemplo de enumeración en C++.

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

3.11. 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 un alcance para identificadores tales como tipos, funciones, variables, etc. Algunas características[2]:

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

[2]Más información: C++ reference: namespace

```
16      cout << ns2::x << '\n';
17  }
18
```

Listing 31. Ejemplo de uso de *namespace*.

3.12. 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.

3.13. Uso de cadenas con *string*

En C++ podemos hacer uso de cadenas por medio del tipo *string*. En compiladortes 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 32. Ejemplo de uso de cadenas *string*.

3.14. 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 33. 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.

3.15. Características relevantes de C++17 [3]

C++17 es la séptima versión del estándar de C++, fue aprobada en diciembre de 2017, y introduce varias características y mejoras en el lenguaje, algunas de las más importantes incluyen:

- **Inicialización de estructuras y clases**: Se permite inicializar estructuras y clases de manera similar a como se inicializan los arreglos y los objetos de cualquier tipo de contenedor. Esto se logra mediante la inclusión de una lista de inicializadores entre llaves {}.
- **Inicialización automática de variables**: Se permite inicializar variables automáticamente en el momento de su declaración, ahorrando la necesidad de escribir código adicional para inicializarlas.
- **Eliminación de if y switch**: Se permite eliminar el cuerpo de un if o un switch si su condición es siempre verdadera o falsa, respectivamente.
- **Programación concurrente y paralela**: Se ha añadido soporte para la programación concurrente y paralela mediante el uso de las librerías estándar de C++.
- **Mejoras en las plantillas**: Se han añadido mejoras en las plantillas, como la posibilidad de especificar un tipo de retorno automático para las funciones plantilla, lo que permite una mayor flexibilidad y legibilidad en el código.
- **Mejoras en la sintaxis**: Se han añadido mejoras en la sintaxis del lenguaje, como la posibilidad de usar el operador .ªuto"para deducir el tipo de una variable en una expresión.

C++17 ha mejorado significativamente el lenguaje, haciéndolo más conciso, fácil de leer y de escribir, y más adecuado para la programación concurrente y paralela. Estas mejoras hacen que el código sea más eficiente y escalable, y que sea más fácil para los desarrolladores escribir código correcto y seguro.

[3] Partes de esta sección fueron desarrollados con apoyo de ChatGPT

3.15.1. Inicialización de estructuras y clases

La inicialización de estructuras y clases es una característica introducida en C++17 que permite inicializar los miembros de una estructura o clase mediante una lista de inicializadores entre llaves {}. Esto es similar a cómo se inicializan los arreglos y los objetos de cualquier tipo de contenedor.

La sintaxis para inicializar una estructura es la siguiente:

```
1
2 struct MyStruct {
3     int x;
4     float y;
5 };
6
7 MyStruct myStruct {1, 3.14f};
8
```

Para una clase:

```
1
2 class MyClass {
3 public:
4     int x;
5     float y;
6 };
7
8 MyClass myClass {1, 3.14f};
9
```

En el ejemplo anterior se define una estructura llamada *MyStruct* con dos miembros: ”x” e ”y”, se declara una variable *myStruct* de ese tipo y se inicializa con los valores 1 y 3.14f respectivamente. Y también se define una clase llamada *MyClass* con dos miembros: ”x” e ”y”, se declara una variable *myClass* de ese tipo y se inicializa con los valores 1 y 3.14f respectivamente.

Además, es posible inicializar solo algunos de los miembros de una estructura o una clase, dejando los demás con su valor por defecto:

```
1 MyStruct myStruct {1};  // x = 1, y = 0.0f (valor por defecto)
2
3 MyClass myClass {1};   // x = 1, y = 0.0f (valor por defecto)
4
```

La inicialización de estructuras y clases es una característica muy útil que permite escribir código más conciso y legible, y también ayuda a prevenir errores comunes relacionados con la inicialización de variables.

3.15.2. Inicialización automática de variables

La inicialización automática de variables es una característica introducida en C++17 que permite inicializar una variable automáticamente en el momento de su declaración, sin necesidad de escribir código adicional para inicializarla. Esto se logra mediante el uso del operador *auto* junto con el valor con el que se desea inicializar la variable.

La sintaxis para inicializar una variable automáticamente es la siguiente:

Sintaxis
`auto <variable>= <valor>;`

Se utiliza el operador *auto* para inicializar automáticamente la variable con el valor, y el compilador deduce automáticamente el tipo de la variable.

Se puede usar auto con cualquier tipo de dato, incluso con tipos compuestos como *structs* o clases:

```
struct MyStruct {
    int x;
    float y;
};

auto myStruct = MyStruct{1, 3.14f};

class MyClass {
public:
    int x;
    float y;
};

auto myClass = MyClass{1, 3.14f};
```

En estos ejemplos, se utiliza el operador *auto* para inicializar automáticamente las variables *myStruct* y *myClass* con los valores especificados en las estructuras *MyStruct* y *MyClass* respectivamente.

La inicialización automática de variables es una característica muy útil que permite escribir código más conciso y legible, ya que el compilador deduce automáticamente el tipo de la variable a partir del valor con el que se inicializa. Esto también ayuda a prevenir errores comunes relacionados con la inicialización de variables, ya que el compilador se encarga de realizar la deducción del tipo de forma segura.

3.15.3. Eliminación de sentencias vacias if y switch

En C++17 se ha introducido la característica de ”Eliminación de sentencias vacías” en las estructuras de control *if* y *switch*.

Se refiere a la eliminación de las sentencias vacías de las estructuras de control *if* y *switch* cuando su condición es siempre verdadera o falsa.

Por ejemplo, consideremos el siguiente código:

```
int x = 5;

if (x > 0)
    x = x + 2;
else
    ; // sentencia vacía

```

En este caso, la condición del *if* es siempre verdadera, por lo que el cuerpo del *else* no se ejecutará nunca. Es decir, la sentencia vacía ";" en el cuerpo del *else* no tiene efecto alguno. El compilador puede eliminarla automáticamente y generar código más eficiente.

Con respecto al *switch*, consideremos el siguiente código:

```
int x = 5;

switch (x) {
    case 5:
        x = x + 2;
        break;
    case 6:
        ; // sentencia vacía
        break;
    default:
        x = x - 2;
}
```

En este caso, el valor de x es siempre 5, por lo que el cuerpo de case 6 no se ejecutará nunca, es decir la sentencia vacía ";"no tiene efecto alguno. El compilador puede eliminarla automáticamente y generar código más eficiente.

La eliminación de sentencias vacías ayuda a escribir código más conciso y legible ya que permite eliminar las estructuras de control innecesarias. Además, también ayuda a mejorar el rendimiento del código ya que el compilador puede generar código más eficiente al eliminar las sentencias vacías.

3.15.4. Mejoras en las plantillas

Las plantillas son una característica de C++ que permite escribir código genérico que se aplica a diferentes tipos de datos. C++17 ha introducido varias mejoras en las plantillas, que hacen que sea más fácil y flexible escribir código genérico.

Una de las mejoras más importantes es la posibilidad de especificar un tipo de retorno automático para las funciones plantilla. Esto permite que el compilador deduzca automáticamente el tipo de retorno de una función plantilla a partir de los argumentos de entrada.

La sintaxis para especificar un tipo de retorno automático para una función plantilla es la siguiente:

```
template <typename T>
auto add(T a, T b) -> decltype(a + b) {
    return a + b;
}

```

En este ejemplo, la función *add* es una plantilla que acepta dos argumentos de cualquier tipo ”T” y devuelve el resultado de la suma de estos argumentos. La función utiliza el operador *decltype* para especificar que el tipo de retorno de la función es el mismo que el tipo de los argumentos de entrada.

Otra mejora importante en las plantillas es la posibilidad de especificar parámetros de plantilla de forma no estricta. Esto significa que se pueden especificar parámetros de plantilla que no son necesariamente tipos.

La sintaxis para especificar parámetros de plantilla no estrictos es la siguiente:

```
template <typename T, int N>
void print_array(T (&arr)[N]) {
    for (int i = 0; i < N; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

En este ejemplo, la función *print_array* es una plantilla que acepta un arreglo *arr* de cualquier tipo ”T” y un entero ”N” que indica el tamaño del arreglo. La función imprime el contenido del arreglo en pantalla.

Estas mejoras en las plantillas hacen que el código sea más fácil y flexible de escribir, ya que permite deducir automáticamente el tipo de retorno y no solo utilizar tipos como parámetros. Esto permite una mayor flexibilidad y legibilidad en el código.

3.15.5. Mejoras en la sintaxis

C++17 ha introducido varias mejoras en la sintaxis del lenguaje que hacen que sea más fácil y legible escribir código. Algunas de las mejoras más importantes son:

Inicializadores de lista en la declaración de variables: Es posible inicializar una variable en el momento de su declaración utilizando una lista de inicializadores entre llaves . Esto es similar a cómo se inicializan los arreglos y los objetos de cualquier tipo de contenedor.

```
int x = {5};
std::vector<int> v = {1, 2, 3};
```

Mejoras en el operador ternario: El operador ternario se ha mejorado para permitir la asignación y el llamado a funciones dentro de su sintaxis.

```
int x = 5;
int y = x > 0 ? x + 2 : x - 2;

std::string message = x > 0 ? "Positive" : "Negative";

```

Mejoras en el operador de propagación: El operador de propagación "..."permite pasar los elementos de un contenedor como argumentos individuales a una función o constructor.

```
std::vector<int> v = {1, 2, 3};
std::initializer_list<int> l = {4, 5, 6};

foo(1, 2, 3);
foo(v.begin(), v.end());
foo(l);

```

Mejoras en el uso de corchetes y llaves: La sintaxis de corchetes y llaves se ha mejorado para permitir su uso como una forma de expresar bloques de código y para inicializar variables.

```
for (int i : {1, 2, 3}) {
    std::cout << i << std::endl;
}

```

Estas mejoras en la sintaxis ayudan a escribir código más legible y fácil de entender, y también ayudan a prevenir errores comunes.

3.15.6. Manejo de cadenas con *string*

Las cadenas de caracteres son elementos fundamentales en la programación, y en C++, se gestionan a través de la clase *std::string*. Esta clase proporciona una interfaz conveniente para manipular cadenas de manera eficiente. A continuación, se presentan un ejemplo completo que ilustran cómo utilizar cadenas en C++.

```
1
2  #include <iostream>
3  #include <string>
4
5  int main() {
6      // Crear una cadena e inicializarla
7      std::string cadena = "Hola, mundo";
8
9      // Imprimir la cadena original
10     std::cout << "Cadena original: " << cadena << std::endl;
11
12     // Obtener la longitud de la cadena
13     size_t longitud = cadena.length();
14     std::cout << "Longitud de la cadena: " << longitud << " caracteres" << std::endl;
15
16     // Acceder a caracteres individuales mediante índices
17     char primerCaracter = cadena[0];
18     char ultimoCaracter = cadena[longitud - 1];
19     std::cout << "Primer caracter: " << primerCaracter << std::endl;
20     std::cout << "Último caracter: " << ultimoCaracter << std::endl;
21
22     // Concatenar cadenas
23     std::string otraCadena = "¡Bienvenido!";
24     std::string concatenacion = cadena + " " + otraCadena;
25     std::cout << "Cadenas concatenadas: " << concatenacion << std::endl;
26
27     // Comparar cadenas
28     std::string otraCadena2 = "Hola, mundo";
29     if (cadena == otraCadena2) {
30         std::cout << "Las cadenas son iguales" << std::endl;
31     } else {
32         std::cout << "Las cadenas son diferentes" << std::endl;
33     }
34
35     std::string cadena1 = "Hola";
36     std::string cadena2 = "Hola";
37     int comparacion = cadena1.compare(cadena2); // comparacion = 0
38     std::cout << "Comparación: "<<comparacion << std::endl;
39
40     // Encontrar subcadenas
41     size_t posicion = cadena.find("mundo");
```

```
42      if (posicion != std::string::npos) {
43          std::cout << "La subcadena 'mundo' comienza en la posición: "
44            << posicion << std::endl;
45      } else {
46          std::cout << "La subcadena 'mundo' no se encontró" << std::endl;
47      }
48
49      // Se pueden convertir cadenas a otros tipos de datos
50      // utilizando métodos como stoi o stof
51      std::string numero = "123";
52      int valor = std::stoi(numero); // valor = 123
53      std::cout << "Valor: " << valor << std::endl;
54
55      std::string decimal = "3.1415";
56      float pi = std::stof(decimal); // pi = 3.1415
57      std::cout << "pi: " << pi << std::endl;
58
59      // Conversión a cadena con el método to_string()
60      cadena = std::to_string(pi);
61      std::cout << cadena << std::endl;
62
63      return 0;
64  }
65
```

Listing 34. Ejemplo de uso de *string* en C++.

En este programa, se ha creado una cadena inicial, se han explorado diversas operaciones comunes de cadenas y se han comentado para proporcionar claridad. Desde la creación e inicialización de cadenas hasta la comparación y búsqueda de subcadenas, estos ejemplos abarcan aspectos esenciales del uso de cadenas en C++. *size_t* es un tipo de datos sin signo en C++ utilizado para representar el tamaño o la longitud de estructuras de datos, como arreglos o cadenas.

Capítulo 4

Introducción a Java

4.1. 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.

4.2. 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.
- 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

[1] Los 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.

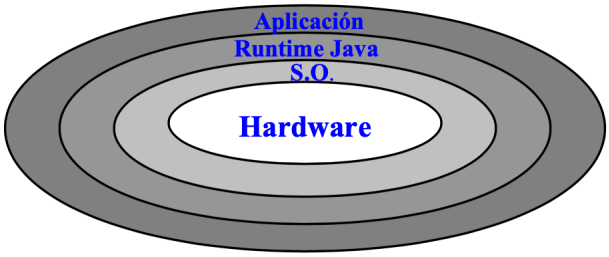


Fig. 4.1. Máquina virtual de Java y su ejecución sobre una plataforma

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 **??**.

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 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 protegidos[2]. 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 referencia 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.

- **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.

4.3. 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.

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

- **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

[3] *Garbage Collector*

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.

4.4. 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*.

4.5. 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, los *Applets*, que son pequeñas aplicaciones escritas en Java, las cuales siguen un conjunto de convenciones que les permiten ejecutarse dentro de un navegador. Estos *applets* siempre están incrustados en una página *html*.

En términos del código fuente las diferencias entre un *applet* y una aplicación son:

- Una aplicación debe definir una clase que contenga el método *main()*, que controla su ejecución. Un *applet* no usa el método *main()*; su ejecución es controlado por varios métodos definidos en la clase *applet*.
- Un *applet*, debe definir una clase derivada de la clase *Applet* [5].

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

[5] A partir de la versión 1.9 del jdk, los applets ya no son soportados.

4.6. 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 (JVM[6]) para la línea de comandos. No dispone de un entorno de desarrollo integrado (IDE), pero es suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones[7].

Los principales programas del *Java Development Kit*:

- **javac**. Es el compilador en línea del JDK.
- **java**. Es la máquina virtual para aplicaciones de Java.
- **appletviewer**. Visor de *applets* de 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
`javac [opciones] <archivo1.java>`

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.

[6] *Java Virtual Machine*

[7] Este 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.

4.7. "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 35. 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 jdk[8]:

```
%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;
```

[8]Se 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.

```
5
6  int main(){
7      cout << "Hola Mundo!" << endl;
8      return 0;
9  }
```

Listing 36. Hola, Mundo en C++.

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[9], podemos ejecutar directamente ejemplos pequeños de código java. Se compila y ejecuta, sin generar el código *.class* correspondiente:

```
% java HolaMundo.java
```

4.8. 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.

4.8.1. 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  }
```

9 10

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/

4.8.2. 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, los vectores 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[11]	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	$3,4*10_{-38}\ldots\ 3,4*10_{38}$
double	real en coma flotante de precisión doble	IEEE 754	64 bits	$1,7*10_{-308}\ldots 1,7*10_{308}$
char	Carácter	Unicode	2 bytes	0 … 65.535
boolean	Lógico		1 bit	true / false

Cuadro 4.1. 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).

4.8.3. 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.

4.8.4. 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 excepciones[12].

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.

4.8.5. 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.

Operadores aritméticos:

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

[12]Este se tocará en otra etapa del curso, al hablar de manejo de excepciones.

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 4.2. Operadores aritméticos Java

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

- ++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--;
```

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

Ejemplo:

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

Cuadro 4.3. Operadores relacionales en Java

```
a = 1; a = 1;
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 **??**

Ejemplo:

```

public class Relacional {
        public static void main(String arg[]) {
                double op1,op2;
                op1=1.34;
                op2=1.35;
                System.out.println("op1="+op1+" op2="+op2);
                System.out.println("op1>op2 = "+(op1>op2));
                System.out.println("op1<op2 = "+(op1<op2));
                System.out.println("op1==op2 = "+(op1==op2));
                System.out.println("op1!=op2 = "+(op1!=op2));
                char op3,op4;
                op3='a'; op4='b';
                System.out.println("'a'>'b' = "+(op3>op4));
        }
}
```

Listing 37. Ejemplo de operadores relacionales en Java.

Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve *true* si son ciertos op1 y op2
||	op1 || op2	O lógico. Devuelve *true* si son ciertos op1 o op2
!	!op1	Negación lógica. Devuelve *true* si es falso op1.

Cuadro 4.4. Operadores lógicos en Java

Operadores lógicos:

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

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 38. Ejemplo de operadores lógicos en Java.

Operadores de asignación:

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

```
op1 = Expresión;
```

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

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 4.5. Operadores de asignación en Java

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 **??**

Precedencia de operadores en Java

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

4.8.6. 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	$[].(parentesis)$
Operadores unarios	$++expr\ --expr - expr\ !$
Creación o conversión de tipo	$new(tipo)expr$
Multiplicación y división	$*/\%$
Suma y resta	$+-$
Desplazamiento de bits	$\ll\gg\ggg>$
Relacionales	$<><=>=$
Igualdad y desigualdad	$==!=$
AND a nivel de bits	&
OR a nivel de bits	$\vert$
AND lógico	&&
OR lógico	$\vert\vert$
Condicional terciaria	? :
Asignación	$=+=-=*=/=\%=\&=\vert=\gg=\ll=\ggg>=$

Cuadro 4.6. 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 **??**

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 4.7. 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.''
```

4.8.7. 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:
`if (<expresión>)`
`  <Bloque instrucciones>`

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 39. 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 40. 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 = nº 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 41. 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 = nº 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 42. 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 = nº 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 43. 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.

Saltos

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 44. 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("Itereación: "+i);
8                  }
9          }
10 }
```

Listing 45. 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.

4.8.8. 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:

```
String Días[]={"Lunes","Martes","Miércoles","Jueves",
               "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:

```
int matriz[][];
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:

```
public class Array1 {
        public static void main (String argumentos[]) {
                String colores[] = {"Rojo","Verde","Azul",
                                    "Amarillo","Negro"};
```

```
                int i;
                for (i=0;i<colores.length;i++)
                        System.out.println(colores[i]);
        }
}
```

Listing 46. Ejemplo de uso de arreglo.

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

```
public class Meses {

        public static void main(String[] args) {
            String meses[] =
               {"Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
               "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"};

            //for(int i = 0; i < meses.length; i++ )
            //   System.out.println("mes: " + meses[i]);

            // sintaxis para recorrer el arreglo y asignar
            // el siguiente elemento a la variable mes en cada ciclo
            // instruccion "for each" a partir de version 5.0 (1.5 del jdk)
            for(String mes: meses)
               System.out.println("mes: " + mes);

        }

}
```

Listing 47. Ejemplo de uso de arreglo 2.

4.8.9. 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:

```
public static final int TEMPO_PRIMAVERA = 0;
public static final int TEMPO_VERANO = 1;
public static final int TEMPO_OTOÑO = 2;
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
`enum <nombreEnum> { <elem 1>, <elem 2>, ..., <elem n> }`

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 48. Ejemplo de enumeración en Java.

4.8.10. 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 linea 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 49. 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 linea desde consola y lectura sin eco (tipo *password*) en la consola.

Ejemplo[13]:

[13]Ejecutar directamente de consola ya que puede no ejecutarse correctamente en el algunos IDEs.

```java
import java.io.Console;

//no funciona en la consola de Eclipse
public class TestConsole {

    public static void main(String... args ) {

            // Obtener un objeto de consola
            Console console = System.console();
            if (console == null) {
            System.err.println("No se obtuvo la consola.");
            System.exit(1);
        }

            String usuario = console.readLine("Usuario:");

            //Lee password y lo recibe en un arreglo de caracteres
            char[] password = console.readPassword("Password: ");

            if (usuario.equals("admin")
             && String.valueOf(password).equals("secreto")) {
                console.printf("Bienvenido %1$s.\n", usuario);

            } else {
                console.printf("Usuario o password inválido.\n");
            }
    }
}
```

Listing 50. 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.

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 51. Ejemplo de entrada de argumentos de cantidad variable.

Otro ejemplo[14], 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
9          for ( double d : numbers )
10            total += d;
11
12         return total / numbers.length;
13      }
14
15      public static void main( String args[] )
16      {
```

[14]Código original de [**?**]

```
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 52. Ejemplo de entrada de argumentos de cantidad variable.

4.8.11. Paquetes

Las clases en Java son organizadas mediante paquetes. Un paquete es entonces el mecanismo para agrupas 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
`package [<ruta>]<nombre paquete>`

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

Capítulo 5

Introducción a Java

5.1. 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.

5.2. 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.
- 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

[1] Los 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.

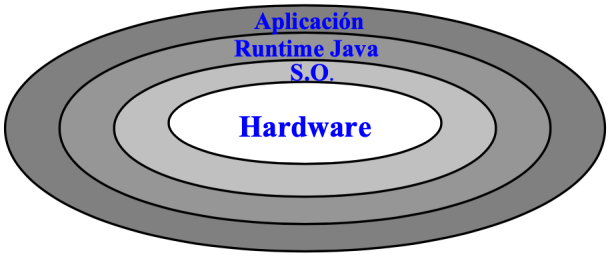


Fig. 5.1. Máquina virtual de Java y su ejecución sobre una plataforma

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 **??**.

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 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 protegidos[2]. 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 referencia 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.

- **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.

5.3. 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.

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

- **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

[3] *Garbage Collector*

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.

5.4. 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*.

5.5. 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, los *Applets*, que son pequeñas aplicaciones escritas en Java, las cuales siguen un conjunto de convenciones que les permiten ejecutarse dentro de un navegador. Estos *applets* siempre están incrustados en una página *html*.

En términos del código fuente las diferencias entre un *applet* y una aplicación son:

- Una aplicación debe definir una clase que contenga el método *main()*, que controla su ejecución. Un *applet* no usa el método *main()*; su ejecución es controlado por varios métodos definidos en la clase *applet*.
- Un *applet*, debe definir una clase derivada de la clase *Applet* [5].

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

[5] A partir de la versión 1.9 del jdk, los applets ya no son soportados.

5.6. 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 (JVM[6]) para la línea de comandos. No dispone de un entorno de desarrollo integrado (IDE), pero es suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones[7].

Los principales programas del *Java Development Kit*:

- **javac**. Es el compilador en línea del JDK.
- **java**. Es la máquina virtual para aplicaciones de Java.
- **appletviewer**. Visor de *applets* de 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
`javac [opciones] <archivo1.java>`

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.

[6] *Java Virtual Machine*

[7] Este 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.

5.7. "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 53. 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 jdk[8]:

```
%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;
```

[8]Se 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.

```
5
6  int main(){
7      cout << "Hola Mundo!" << endl;
8      return 0;
9  }
```

Listing 54. Hola, Mundo en C++.

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[9], podemos ejecutar directamente ejemplos pequeños de código java. Se compila y ejecuta, sin generar el código *.class* correspondiente:

```
% java HolaMundo.java
```

5.8. 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.

5.8.1. 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  }
```

9 10

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/

5.8.2. 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, los vectores 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[11]	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	$3,4*10_{-38}\ldots\ 3,4*10_{38}$
double	real en coma flotante de precisión doble	IEEE 754	64 bits	$1,7*10_{-308}\ldots 1,7*10_{308}$
char	Carácter	Unicode	2 bytes	0 … 65.535
boolean	Lógico		1 bit	true / false

Cuadro 5.1. 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).

5.8.3. 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.

5.8.4. 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 excepciones[12].

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.

5.8.5. 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.

Operadores aritméticos:

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

[12]Este se tocará en otra etapa del curso, al hablar de manejo de excepciones.

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 5.2. Operadores aritméticos Java

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

- ++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--;
```

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

Ejemplo:

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

Cuadro 5.3. Operadores relacionales en Java

```
a = 1; a = 1;
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 **??**

Ejemplo:

```

public class Relacional {
        public static void main(String arg[]) {
                double op1,op2;
                op1=1.34;
                op2=1.35;
                System.out.println("op1="+op1+" op2="+op2);
                System.out.println("op1>op2 = "+(op1>op2));
                System.out.println("op1<op2 = "+(op1<op2));
                System.out.println("op1==op2 = "+(op1==op2));
                System.out.println("op1!=op2 = "+(op1!=op2));
                char op3,op4;
                op3='a'; op4='b';
                System.out.println("'a'>'b' = "+(op3>op4));
        }
}
```

Listing 55. Ejemplo de operadores relacionales en Java.

Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve *true* si son ciertos op1 y op2
||	op1 || op2	O lógico. Devuelve *true* si son ciertos op1 o op2
!	!op1	Negación lógica. Devuelve *true* si es falso op1.

Cuadro 5.4. Operadores lógicos en Java

Operadores lógicos:

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

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 56. Ejemplo de operadores lógicos en Java.

Operadores de asignación:

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

```
op1 = Expresión;
```

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

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 5.5. Operadores de asignación en Java

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 **??**

Precedencia de operadores en Java

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

5.8.6. 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	$[].(parentesis)$
Operadores unarios	$++expr\ --expr - expr\ !$
Creación o conversión de tipo	$new(tipo)expr$
Multiplicación y división	$*/\%$
Suma y resta	$+-$
Desplazamiento de bits	$\ll\gg\ggg>$
Relacionales	$<><=>=$
Igualdad y desigualdad	$==!=$
AND a nivel de bits	&
OR a nivel de bits	$\vert$
AND lógico	&&
OR lógico	$\vert\vert$
Condicional terciaria	? :
Asignación	$=+=-=*=/=\%=\&=\vert=\gg=\ll=\ggg>=$

Cuadro 5.6. 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 **??**

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 5.7. 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.''
```

5.8.7. 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:

```
if (<expresión>)
    <Bloque instrucciones>
```

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 57. 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 58. 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 = nº 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 59. 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 = nº 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 60. 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 = nº 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 61. 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.

Saltos

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 62. 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("Itereación: "+i);
8                  }
9          }
10 }
```

Listing 63. 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.

5.8.8. 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:

```
String Días[]={"Lunes","Martes","Miércoles","Jueves",
               "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:

```
int matriz[][];
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:

```
public class Array1 {
        public static void main (String argumentos[]) {
                String colores[] = {"Rojo","Verde","Azul",
                                    "Amarillo","Negro"};
```

```
5                   int i;
6                   for (i=0;i<colores.length;i++)
7                           System.out.println(colores[i]);
8           }
9   }
```

Listing 64. 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 65. Ejemplo de uso de arreglo 2.

5.8.9. 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_OTOÑO = 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
`enum <nombreEnum> { <elem 1>, <elem 2>, ..., <elem n> }`

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 66. Ejemplo de enumeración en Java.

5.8.10. 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 linea 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 67. 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 linea desde consola y lectura sin eco (tipo *password*) en la consola.

Ejemplo[13]:

[13]Ejecutar directamente de consola ya que puede no ejecutarse correctamente en el algunos IDEs.

```java
import java.io.Console;

//no funciona en la consola de Eclipse
public class TestConsole {

    public static void main(String... args ) {

            // Obtener un objeto de consola
            Console console = System.console();
            if (console == null) {
            System.err.println("No se obtuvo la consola.");
            System.exit(1);
        }

            String usuario = console.readLine("Usuario:");

            //Lee password y lo recibe en un arreglo de caracteres
            char[] password = console.readPassword("Password: ");

            if (usuario.equals("admin")
             && String.valueOf(password).equals("secreto")) {
                console.printf("Bienvenido %1$s.\n", usuario);

            } else {
                console.printf("Usuario o password inválido.\n");
            }
    }
}
```

Listing 68. 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.

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 69. Ejemplo de entrada de argumentos de cantidad variable.

Otro ejemplo[14], 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
9          for ( double d : numbers )
10            total += d;
11
12         return total / numbers.length;
13      }
14
15      public static void main( String args[] )
16      {
```

[14]Código original de [**?**]

```
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 70. Ejemplo de entrada de argumentos de cantidad variable.

5.8.11. Paquetes

Las clases en Java son organizadas mediante paquetes. Un paquete es entonces el mecanismo para agrupas 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
`package [<ruta>]<nombre paquete>`

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

Capítulo 6

Introducción a Ruby

6.1. 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 Phyton, 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.

Ruby's architect,
Yukihiro Matsumoto

Matsumoto dice haber puesto énfasis en crear un lenguaje productivo y divertido, siguiendo los principios de buen diseñó 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.

6.2. 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*

6.2.1. 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 *byte-code* 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.

6.2.2. 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.

6.2.3. 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 $foo = Foo.new(\text{"}hola\text{"})$ en vez de $Foo\,foo = newFoo(\text{"}hola\text{"})$.

- 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*.

6.3. Herramientas

Existen dos herramientas básicas en Ruby:

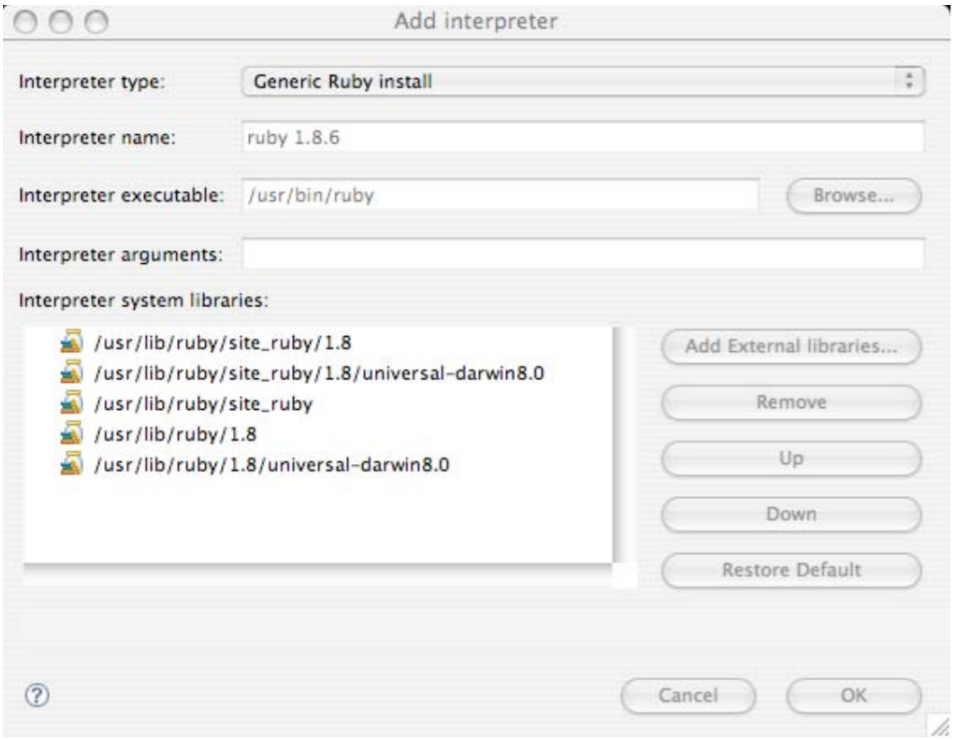
- **ruby**. Es el intérprete del lenguaje. Puede recibir expresiones del lenguaje como parámetros o archivos con programas.

- **irb** (o fxri en algunas versiones para Windows). Este es Ruby interactivo (*Interactive RuBy*) que permite recibir expresiones del lenguaje e irlas interpretando línea por línea, como cualquier lenguaje interpretado. Además:

- **ri**. Documentación de clases estándar de ruby.

También es posible integrar al interprete al IDE de Eclipse. Para esto se debe agregar el plugin llamado *Ruby Development Tools* (RDT) aparte de tener instalado el interprete en la computadora. El plugin puede ser encontrado en: http://rubyeclipse.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:



6.4. Ruby: Fundamentos del lenguaje

6.4.1. 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 $\backslash$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:

```
=begin
Este es un comentario
=end
```

Identificadores

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

6.4.2. Literales

Enteros

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

```
123                     # decimal
0377                    # octal
0xff                    # hexadecimal
0b1011                  # binary
?a                      # código para 'a'
12345678901234567890    # Bignum: entero de longitud infinita
```

Flotantes

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

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

Cadenas

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

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

Concatenación

Cadenas adyacentes son concatenadas :

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

6.4.3. 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 predefinidas 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*.

6.4.4. Operadores

Asignación

La asignación función 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.

6.4.5. Operador ternario

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

```
a? b : c
```

6.4.6. 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 &&
```

```
||
.. ...    # Operadores de rango (... excluyendo el límite derecho)
?:
=   # (y operadores abreviados como +=, -=, etc.)
not
and or
```

Operadores que no pueden ser redefinidos

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

```
...
!
not
&&
and
||
or
::
=
+=, -=,  # (y el resto de las asignaciones abreviadas)
? :
```

6.4.7. 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 $\ll$. Ruby dinámicamente ajusta el tamaño del arreglo al añadir o remover los elementos:

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

El operador $\ll$ 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:

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

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

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

a[0]                                  # => 1
```

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

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

El tamaño del arreglo es validado:

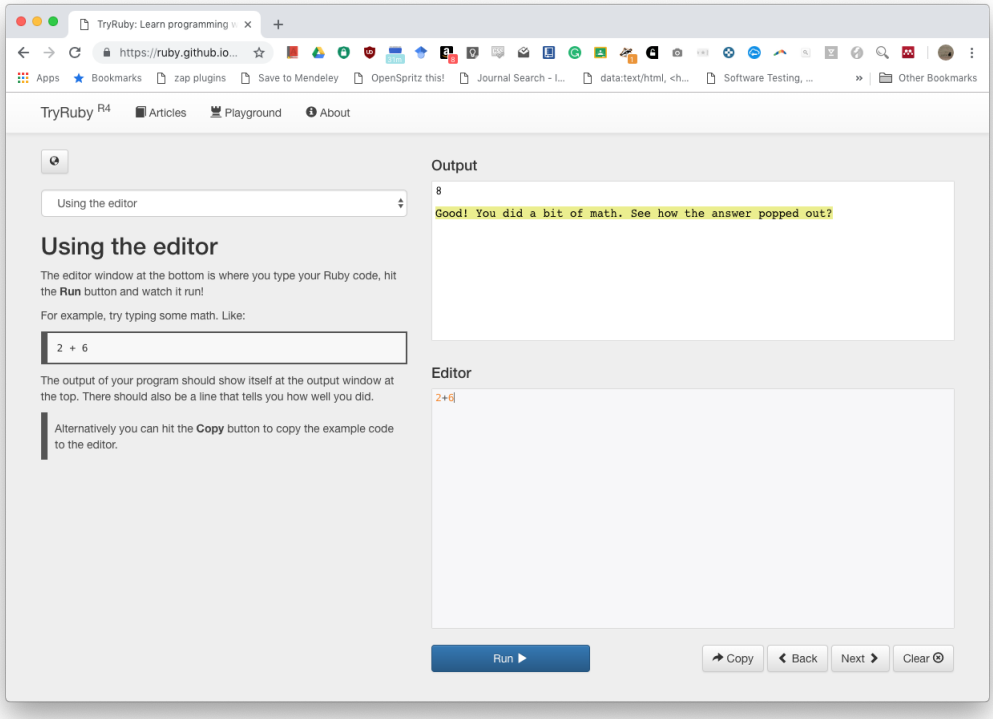
```
a[5]           # => nil
```

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:

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

6.4.8. Probando Ruby

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



6.4.9. 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 clausula *else* se ejecuta en el caso de que ningún segmento *when* sea ejecutado.

Sintaxis

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

Ejemplo[1]:

```
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 ';'.

[1]Recordar 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 entre las clausulas *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

```
until condicional [do]
  instrucciones
end
```

o:

```
instrucciones until condicional
```

o:

```
begin
  instrucciones
end until condicional
```

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

```
for variable in expresion [do]
  instrucciones
end
```

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.

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

Listing 71. Ejemplo *redo* en Ruby.

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

```

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

Listing 72. 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
```

```
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break unless i <= n
end

# Ciclo 7 (for)
n=list.size-1
for i in 0..n do
  print "#{list[i]} "
end

10.times do
  puts "hello"
end
Programa 37
```

Listing 73. Ejemplos ciclos en Ruby.

```

car = "Patriot"

manufacturer = case car
   when "Focus" then "Ford"
   when "Navigator"
        "Lincoln"
   when "Camry"
        "Toyota"
   when "Civic"
        "Honda"
   when "Patriot" then "Jeep"
   when "Jetta" then "VW"
   when "Ceyene" then "Porsche"
   when "Outback"
        "Subaru"
   when "520i": "BMW"
   when "Tundra": "Nissan"
   else "Desconocido"
end

```

```
22 puts "El " + car  + " es fabricado por "  + manufacturer
```

Listing 74. 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 75. 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 76. Ejemplo de *for* en Ruby.

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

Listing 77. Ejemplo 2 de *for* en Ruby.

6.4.10. 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 similas 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 78. 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 79. Ejemplo de entrada simple en Ruby.

6.4.11. Módulos en Ruby

Un **módulo** es definido con la sintaxis:

Sintaxis

```
module <nombre-módulo>
  <código>
end
```

De hecho, un módulo, aunque es similar a una clase no puede tener instancias ni subclases y añade un nuevo alcance para variables locales. Un módulo que es definido con el nombre de otro previamente definido añadirá sus definiciones al módulo inicial. En el diseño de Ruby, la clase *Module* de *module* es la superclase de la clase *Class* de *class* ! [**?**].

Cuando existe ambigüedad, es posible referirse a un método o identificador dentro de un módulo usando el operador ::, por ejemplo:

```
nombre-módulo::método
```

También es posible referirse directamente a los elementos dentro de un módulo sin necesidad de usar el nombre del módulo y el operador :: en cada ocasión. Para esto podemos incluir (*include*) el módulo. Esta característica da además lugar a los *mixins* que se verán posteriormente. Es importante notar que *include* hace referencia a un módulo y no a un archivo. Si el módulo esta en un diferente archivo, este debe ser solicitado mediante la instrucción *require* antes de poder ser incluido [**?**].

Capítulo 7

Introducción a Ruby

7.1. 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 Phyton, 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.


Ruby's architect,
Yukihiro Matsumoto

Matsumoto dice haber puesto énfasis en crear un lenguaje productivo y divertido, siguiendo los principios de buen diseñó 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.

7.2. 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*

7.2.1. 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 *byte-code* 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.

7.2.2. 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.

7.2.3. 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 $foo = Foo.new(\text{"}hola\text{"})$ en vez de $Foo\,foo = newFoo(\text{"}hola\text{"})$.

- 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*.

7.3. Herramientas

Existen dos herramientas básicas en Ruby:

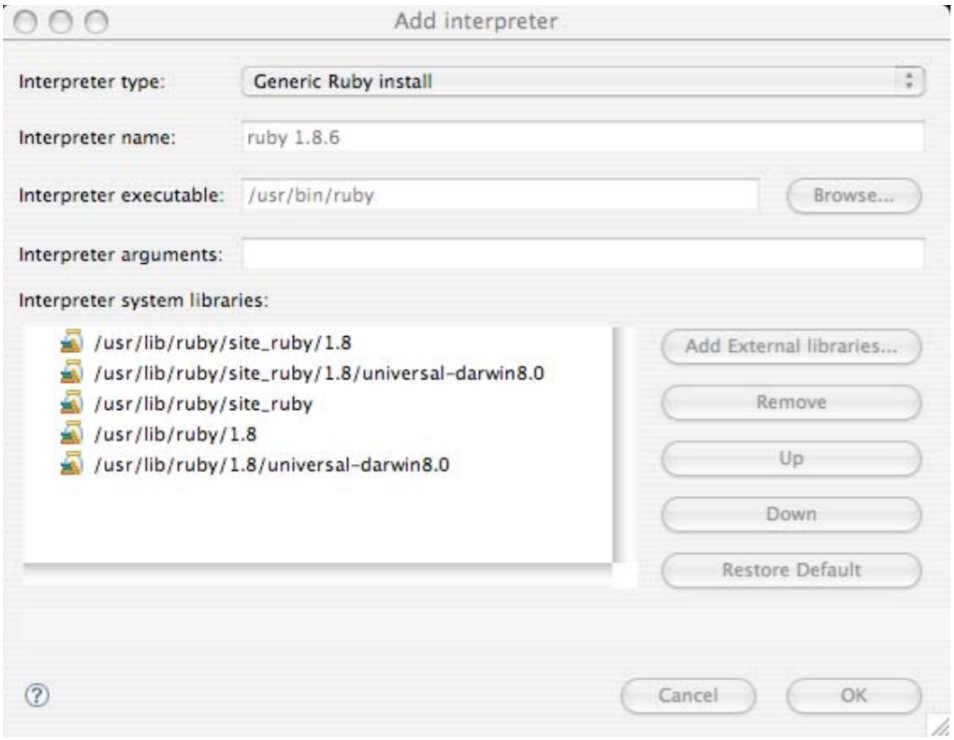
- **ruby**. Es el intérprete del lenguaje. Puede recibir expresiones del lenguaje como parámetros o archivos con programas.

- **irb** (o fxri en algunas versiones para Windows). Este es Ruby interactivo (*Interactive RuBy*) que permite recibir expresiones del lenguaje e irlas interpretando línea por línea, como cualquier lenguaje interpretado. Además:

- **ri**. Documentación de clases estándar de ruby.

También es posible integrar al interprete al IDE de Eclipse. Para esto se debe agregar el plugin llamado *Ruby Development Tools* (RDT) aparte de tener instalado el interprete en la computadora. El plugin puede ser encontrado en: http://rubyeclipse.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:



7.4. Ruby: Fundamentos del lenguaje

7.4.1. 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 $\backslash$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:

```
=begin
Este es un comentario
=end
```

Identificadores

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

7.4.2. Literales

Enteros

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

```
123                     # decimal
0377                    # octal
0xff                    # hexadecimal
0b1011                  # binary
?a                      # código para 'a'
12345678901234567890    # Bignum: entero de longitud infinita
```

Flotantes

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

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

Cadenas

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

```
"una cadena"    # permite sustitución y notación con \
‘otra cadena’     # no permite sustitución y solo \ \ ó \’.
```

Concatenación

Cadenas adyacentes son concatenadas :

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

7.4.3. 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 predefinidas 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*.

7.4.4. Operadores

Asignación

La asignación función 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.

7.4.5. Operador ternario

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

```
a? b : c
```

7.4.6. 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 &&
```

```
||
.. ...    # Operadores de rango (... excluyendo el límite derecho)
?:
=   # (y operadores abreviados como +=, -=, etc.)
not
and or
```

Operadores que no pueden ser redefinidos

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

```
...
!
not
&&
and
||
or
::
=
+=, -=,  # (y el resto de las asignaciones abreviadas)
? :
```

7.4.7. 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 $\ll$. Ruby dinámicamente ajusta el tamaño del arreglo al añadir o remover los elementos:

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

El operador $\ll$ 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:

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

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

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

a[0]                                  # => 1
```

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

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

El tamaño del arreglo es validado:

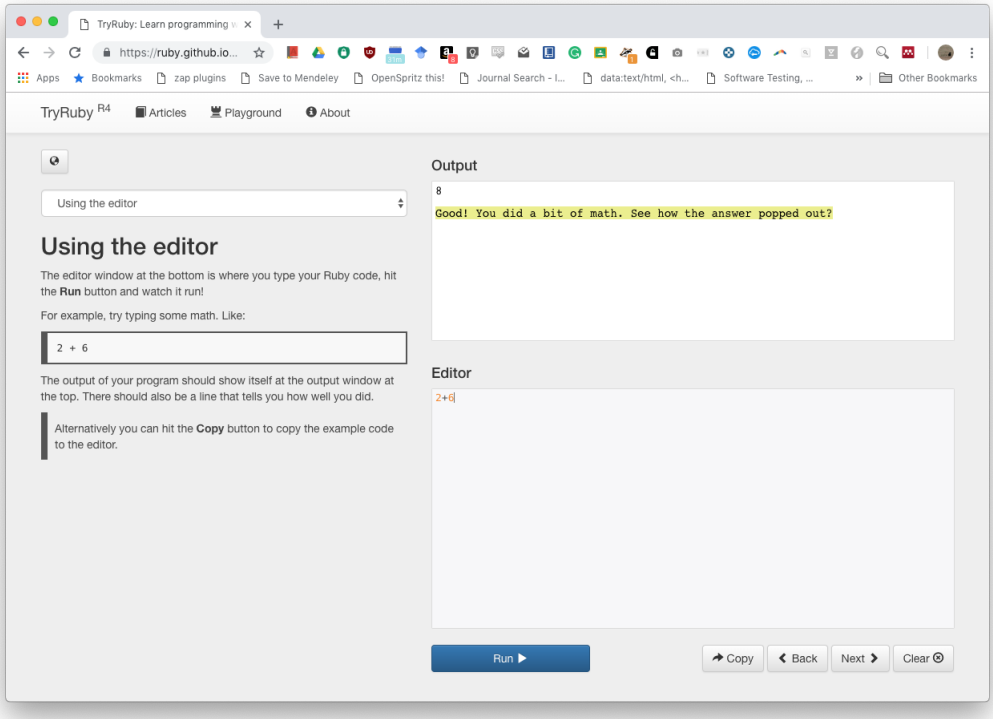
```
a[5]           # => nil
```

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:

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

7.4.8. Probando Ruby

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



7.4.9. 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 clausula *else* se ejecuta en el caso de que ningún segmento *when* sea ejecutado.

Sintaxis

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

Ejemplo[1]:

```
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 ';'.

[1]Recordar 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 entre las clausulas *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

```
until condicional [do]
  instrucciones
end
```

o:

```
instrucciones until condicional
```

o:

```
begin
  instrucciones
end until condicional
```

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

```
for variable in expresion [do]
  instrucciones
end
```

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.

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

Listing 80. Ejemplo *redo* en Ruby.

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

```

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

Listing 81. 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
```

```
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break unless i <= n
end

# Ciclo 7 (for)
n=list.size-1
for i in 0..n do
  print "#{list[i]} "
end

10.times do
  puts "hello"
end
Programa 37
```

Listing 82. Ejemplos ciclos en Ruby.

```

car = "Patriot"

manufacturer = case car
   when "Focus" then "Ford"
   when "Navigator"
        "Lincoln"
   when "Camry"
        "Toyota"
   when "Civic"
        "Honda"
   when "Patriot" then "Jeep"
   when "Jetta" then "VW"
   when "Ceyene" then "Porsche"
   when "Outback"
        "Subaru"
   when "520i": "BMW"
   when "Tundra": "Nissan"
   else "Desconocido"
end

```

```
22 puts "El " + car  + " es fabricado por "  + manufacturer
```

Listing 83. 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 84. 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 85. Ejemplo de *for* en Ruby.

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

Listing 86. Ejemplo 2 de *for* en Ruby.

7.4.10. 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 similas 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 87. 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 88. Ejemplo de entrada simple en Ruby.

7.4.11. Módulos en Ruby

Un **módulo** es definido con la sintaxis:

Sintaxis

```
module <nombre-módulo>
  <código>
end
```

De hecho, un módulo, aunque es similar a una clase no puede tener instancias ni subclases y añade un nuevo alcance para variables locales. Un módulo que es definido con el nombre de otro previamente definido añadirá sus definiciones al módulo inicial. En el diseño de Ruby, la clase *Module* de *module* es la superclase de la clase *Class* de *class* ! [**?**].

Cuando existe ambigüedad, es posible referirse a un método o identificador dentro de un módulo usando el operador ::, por ejemplo:

```
nombre-módulo::método
```

También es posible referirse directamente a los elementos dentro de un módulo sin necesidad de usar el nombre del módulo y el operador :: en cada ocasión. Para esto podemos incluir (*include*) el módulo. Esta característica da además lugar a los *mixins* que se verán posteriormente. Es importante notar que *include* hace referencia a un módulo y no a un archivo. Si el módulo esta en un diferente archivo, este debe ser solicitado mediante la instrucción *require* antes de poder ser incluido [**?**].

Capítulo 8

Introducción a Python

8.1. 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 Pyhton:

- *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 ejecución[2].
- *Jython*. Es una implementación de Pyhton 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

8.2. 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

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

[2] Ver: python.org

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.

8.3. Fundamentos de Python

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

8.3.2. 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.

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 8.1. Operadores aritméticos en Python

8.3.3. 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*.

8.3.4. Operadores

Operadores aritméticos

Los operadores aritméticos que son utilizados en Python son mostrados en el Cuadro **??**

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

```
1 >>>monto_bruto = 175
2 >>>tasa_interes = 12
```

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 8.2. Operadores relacionales en Python

```
>>> monto_interes = monto_bruto * tasa_interes / 100
>>> tasa_bonificacion = 5
>>> importe_bonificacion = monto_bruto * tasa_bonificacion / 100
>>> monto_neto = (monto_bruto - importe_bonificacion) + monto_interes
>>> monto_neto
187.25
```

Operadores relacionales

Ver Cuadro **??** con el listado de los operadores relacionales.

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

```
>>> 5>3>10
False
>>> 5>10>1
False
>>> 5>3>2
True
>>> 5>2>3
False
>>> 5>2<3
True
```

Operadores lógicos

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

8.3.5. Tipos de datos

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

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 8.3. Operadores lógicos en Python

```
Cadena de texto (String)
mi_cadena = "Hola mundo"
Número entero
edad = 35
Número hexadecimal
edad = 0x23
Número real
precio = 7435.28
Booleano (verdadero / falso)
verdadero = True
        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.

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

También pueden agregarse nuevos datos a la lista,

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

Ejemplos:

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

>>> lista=lista+[4,5,6]
>>> lista
[123, 'xxx', 3.14, 4, 5, 6]
>>> lista.append('zzz')
>>> lista
[123, 'xxx', 3.14, 4, 5, 6, 'zzz']
>>> lista.pop(2)
3.14
>>> lista
[123, 'xxx', 4, 5, 6, 'zzz']
>>> orden=['c', 'a', 'b']
>>> orden.sort()
>>> orden
['a', 'b', 'c']
>>> orden.reverse()
>>> orden
['c', 'b', 'a']
>>> lista
[123, 'xxx', 4, 5, 6, 'zzz']
>>> lista[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> lista[100]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range


```

```
>>> matriz=[[1, 2, 3],
... [4, 5, 6],
... [7, 8, 9]]
>>> matriz
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matriz[1]
[4, 5, 6]
>>> matriz[1][2]
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.

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

Conjuntos

Podemos también crear un conjunto

```
>>> s={1, 2, 3}
>>> print(s)
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(2)
>>> s
{1, 2, 3, 4}
>>> s.discard(2)
>>> s
{1, 3, 4}
>>> s.remove(4)
>>> s
{1, 3}
>>> s.discard(4)
>>> s.remove(4)
```

```
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    s.remove(4)
KeyError: 4
```

Diccionarios

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

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

Un diccionario permite eliminar cualquier elemento,

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

Al igual que las listas permite modificar los elementos,

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

8.3.6. 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:

```

        >>> def cruzar(semaforo):
        ...             if semaforo == "verde":
        ...                         print ("Cruzar la calle")
        ...             else:
```

```
...                         print ("No cruzar la calle")

>>>def decidir_pago (compra)
...         if compra <= 100:
...                          print ("Pago en efectivo")
...          elif compra > 100 and compra < 300:
...                          print ("Pago con tarjeta de débito")
...          else:
...                          print ("Pago con tarjeta de crédito")
```

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:

```
lang = input("¿Qué lenguaje de programación quieres aprender? ")
match lang:
    case "JavaScript":
        print("Puedes convertirte en un desarrollador web.")
    case "Python":
        print("Puedes convertirte en un científico de datos")
    case "PHP":
        print("Puedes convertirte en un desarrollador backend")
    case "Solidity":
        print("Puedes convertirte en un desarrollador de Blockchain")
    case "Java":
        print("Puedes convertirte en un desarrollador de aplicaciones móviles")
    case _:
        print("El lenguaje no importa, lo que importa es resolver problemas.")
```

Listing 89. 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:

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

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

```
>>> imp_anios(2009)
Informes del año 2009
Informes del año 2010
Informes del año 2011
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:

```
>>> def edad():
        while True:
                edad = input("Edad: ")
                if int(edad)>100:
                        break

>>> edad()
Edad: 4
Edad: 5
Edad: 101
>>>
```

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:

```
>>> mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
        >>> for nombre in mi_lista:
        ...     print nombre
        ...

```

Este programa devuelve como resultado:

```
 Juan
     Antonio
     Pedro
     Herminio
```

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

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

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.

8.3.7. 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. $print(\text{"}Hola\text{"}, alguien, \text{"!"})$.
- 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.

[3] Antes de Python 3 era llamada raw_input

```
>>> variable=input("Edad:")
Edad:45
>>> variable
'45'
>>> type(variable)
<class 'str'>
>>> int(variable)
45

>>> nombre=input("Nombre:")
Nombre:carlos a
>>> nombre
'carlos a'
```

Ejemplo:

```
s=" ¡Hola, Mundo! "
print(s)
print(s[1])
print(s[7:12])

# elimina caracteres en blanco del lado izquierdo y derecho de la cadena
print(s.strip())
print(len(s))
print(s.lower())
print(s.upper())

#sustituye "l" por "j"
print(s.replace("l", "j"))

#separa una cadena y regresa una lista de cadenas
str = "Un ejemplo de string....!"
print (str.split( ))
print (str.split('e',1))
print (str.split('e'))

print("Nombre:")
n=input()
print("Hola, "+n)


```

Listing 90. Ejemplo de entrada en Python.

8.3.8. 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 91. 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 92. Ejemplo en Python.

8.3.9. 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 "funcionz 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 esta 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.

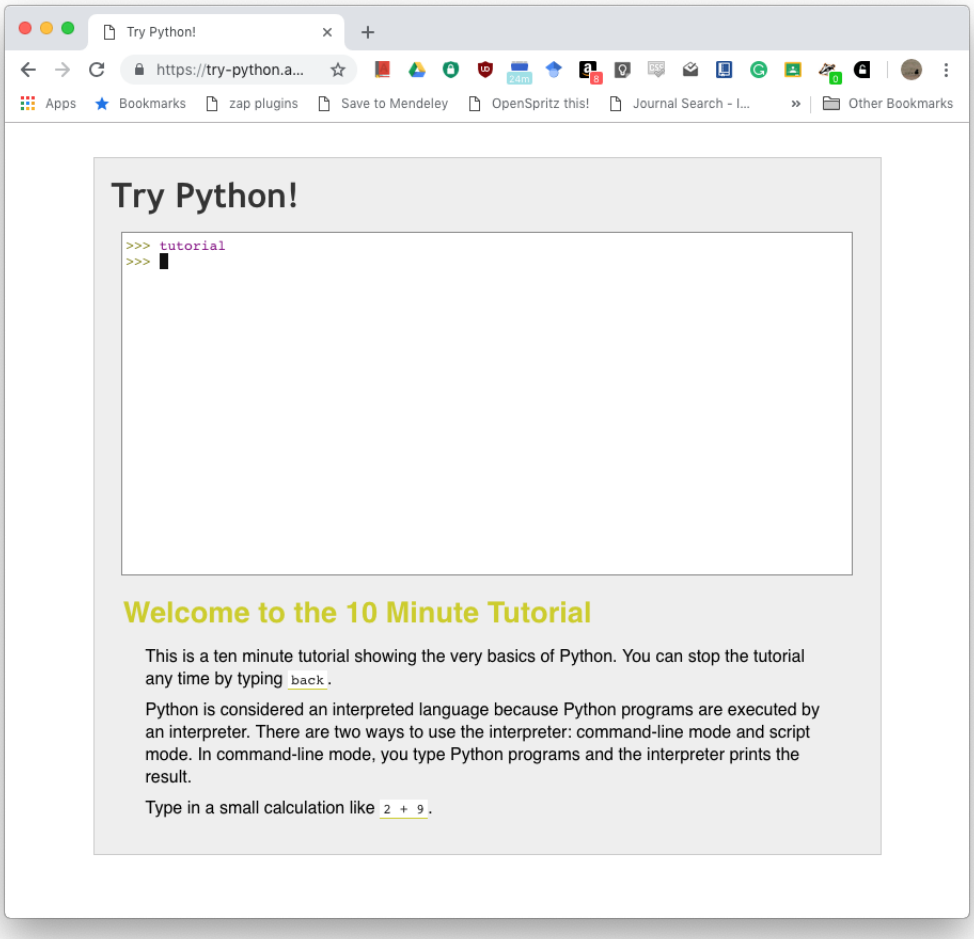
8.3.10. Paquetes en python

Un paquete es una carpeta que contiene archivos *.py*. Pero para que una carpeta pueda ser considerada un paquete, esta debe contener una 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.

8.3.11. Probando Pyhton

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



8.3.12. Pyhton 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 esta 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 9

Introducción a Python

9.1. 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 Pyhton:

- *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 ejecución[2].
- *Jython*. Es una implementación de Pyhton 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

9.2. 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

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

[2] Ver: python.org

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.

9.3. Fundamentos de Python

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

9.3.2. 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.

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 9.1. Operadores aritméticos en Python

9.3.3. 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*.

9.3.4. Operadores

Operadores aritméticos

Los operadores aritméticos que son utilizados en Python son mostrados en el Cuadro **??**

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

```
1 >>>monto_bruto = 175
2 >>>tasa_interes = 12
```

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 9.2. Operadores relacionales en Python

```
>>> monto_interes = monto_bruto * tasa_interes / 100
>>> tasa_bonificacion = 5
>>> importe_bonificacion = monto_bruto * tasa_bonificacion / 100
>>> monto_neto = (monto_bruto - importe_bonificacion) + monto_interes
>>> monto_neto
187.25
```

Operadores relacionales

Ver Cuadro **??** con el listado de los operadores relacionales.

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

```
>>> 5>3>10
False
>>> 5>10>1
False
>>> 5>3>2
True
>>> 5>2>3
False
>>> 5>2<3
True
```

Operadores lógicos

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

9.3.5. Tipos de datos

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

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 9.3. Operadores lógicos en Python

```
Cadena de texto (String)
mi_cadena = "Hola mundo"
Número entero
edad = 35
Número hexadecimal
edad = 0x23
Número real
precio = 7435.28
Booleano (verdadero / falso)
verdadero = True
        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.

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

También pueden agregarse nuevos datos a la lista,

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

Ejemplos:

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

>>> lista=lista+[4,5,6]
>>> lista
[123, 'xxx', 3.14, 4, 5, 6]
>>> lista.append('zzz')
>>> lista
[123, 'xxx', 3.14, 4, 5, 6, 'zzz']
>>> lista.pop(2)
3.14
>>> lista
[123, 'xxx', 4, 5, 6, 'zzz']
>>> orden=['c', 'a', 'b']
>>> orden.sort()
>>> orden
['a', 'b', 'c']
>>> orden.reverse()
>>> orden
['c', 'b', 'a']
>>> lista
[123, 'xxx', 4, 5, 6, 'zzz']
>>> lista[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> lista[100]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range


```

```
>>> matriz=[[1, 2, 3],
... [4, 5, 6],
... [7, 8, 9]]
>>> matriz
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matriz[1]
[4, 5, 6]
>>> matriz[1][2]
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.

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

Conjuntos

Podemos también crear un conjunto

```
>>> s={1, 2, 3}
>>> print(s)
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(2)
>>> s
{1, 2, 3, 4}
>>> s.discard(2)
>>> s
{1, 3, 4}
>>> s.remove(4)
>>> s
{1, 3}
>>> s.discard(4)
>>> s.remove(4)
```

```
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    s.remove(4)
KeyError: 4
```

Diccionarios

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

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

Un diccionario permite eliminar cualquier elemento,

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

Al igual que las listas permite modificar los elementos,

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

9.3.6. 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:

```

        >>> def cruzar(semaforo):
        ...             if semaforo == "verde":
        ...                         print ("Cruzar la calle")
        ...             else:
```

```
...                         print ("No cruzar la calle")

>>>def decidir_pago (compra)
...         if compra <= 100:
...                          print ("Pago en efectivo")
...          elif compra > 100 and compra < 300:
...                          print ("Pago con tarjeta de débito")
...          else:
...                          print ("Pago con tarjeta de crédito")
```

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:

```
lang = input("¿Qué lenguaje de programación quieres aprender? ")
match lang:
    case "JavaScript":
        print("Puedes convertirte en un desarrollador web.")
    case "Python":
        print("Puedes convertirte en un científico de datos")
    case "PHP":
        print("Puedes convertirte en un desarrollador backend")
    case "Solidity":
        print("Puedes convertirte en un desarrollador de Blockchain")
    case "Java":
        print("Puedes convertirte en un desarrollador de aplicaciones móviles")
    case _:
        print("El lenguaje no importa, lo que importa es resolver problemas.")
```

Listing 93. 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:

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

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

```
>>> imp_anios(2009)
Informes del año 2009
Informes del año 2010
Informes del año 2011
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:

```
>>> def edad():
        while True:
                edad = input("Edad: ")
                if int(edad)>100:
                        break

>>> edad()
Edad: 4
Edad: 5
Edad: 101
>>>
```

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:

```
>>> mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
        >>> for nombre in mi_lista:
        ...     print nombre
        ...

```

Este programa devuelve como resultado:

```
 Juan
     Antonio
     Pedro
     Herminio
```

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

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

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.

9.3.7. 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. $print(\text{"}Hola\text{"}, alguien, \text{"!"})$.
- 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.

[3] Antes de Python 3 era llamada raw_input

```
>>> variable=input("Edad:")
Edad:45
>>> variable
'45'
>>> type(variable)
<class 'str'>
>>> int(variable)
45

>>> nombre=input("Nombre:")
Nombre:carlos a
>>> nombre
'carlos a'
```

Ejemplo:

```
s=" ¡Hola, Mundo! "
print(s)
print(s[1])
print(s[7:12])

# elimina caracteres en blanco del lado izquierdo y derecho de la cadena
print(s.strip())
print(len(s))
print(s.lower())
print(s.upper())

#sustituye "l" por "j"
print(s.replace("l", "j"))

#separa una cadena y regresa una lista de cadenas
str = "Un ejemplo de string....!"
print (str.split( ))
print (str.split('e',1))
print (str.split('e'))

print("Nombre:")
n=input()
print("Hola, "+n)


```

Listing 94. Ejemplo de entrada en Python.

9.3.8. 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 95. 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 96. Ejemplo en Python.

9.3.9. 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 "funcionz 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 esta 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.

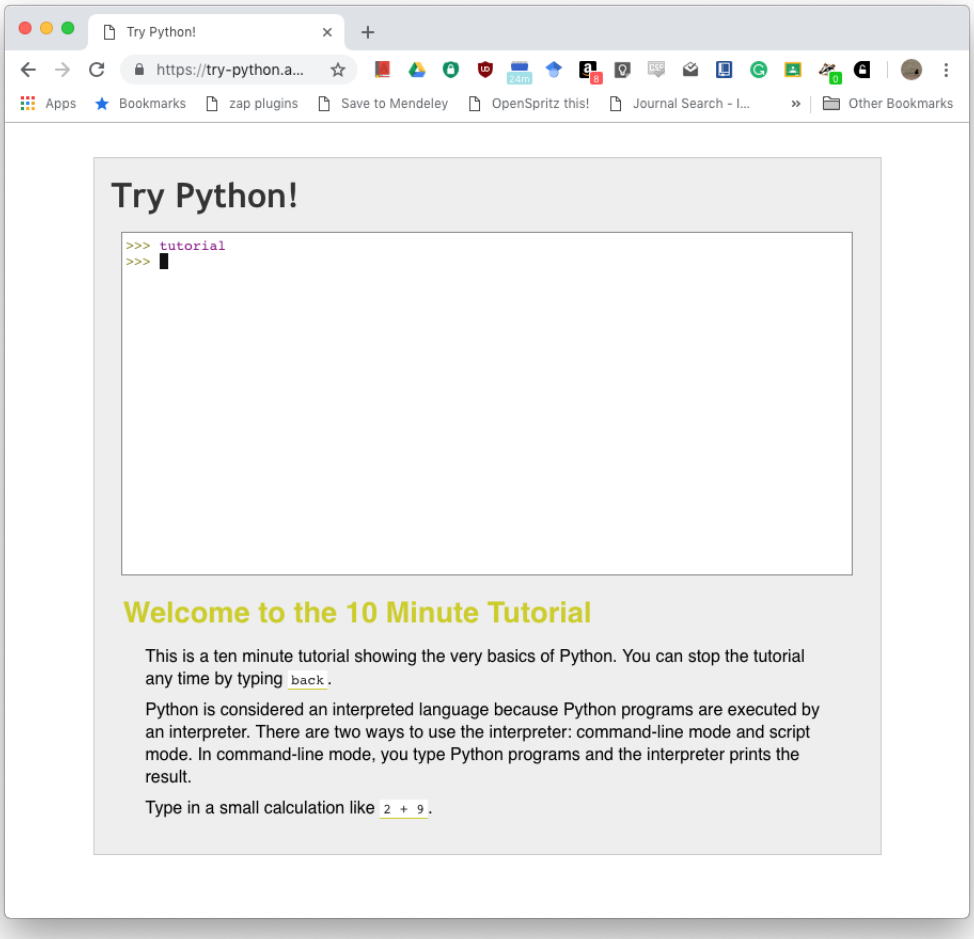
9.3.10. Paquetes en python

Un paquete es una carpeta que contiene archivos *.py*. Pero para que una carpeta pueda ser considerada un paquete, esta debe contener una 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.

9.3.11. Probando Pyhton

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



9.3.12. Pyhton 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 esta 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 10

Introducción a C#

10.1. Introducción

C# (C *Sharp*) fue presentado en el verano del 2000 en su versión 1.0, junto con el framework .NET por Microsoft. Aunque le 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 lenguaje[1]. 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 lider del proyecto de C#) no esta 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 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 adquiriendo fuerza con el tiempo y se ha constituido en una opción real de desarrollo de software desde hace un buen tiempo.

[1] Programa 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).

10.2. 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 *MonoDevelop*. 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

10.3. Fundamentos de C#

10.3.1. 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

10.3.2. 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*.

10.3.3. 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.

10.3.4. 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

10.3.5. 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) { ... }
```

10.3.6. 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 97. 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                         Console.WriteLine();
11                         Console.Write("Leyendo entrada con Read(): ");
```

```
12                          entradaUsuario = Console.Read();
13                          Console.WriteLine("Valor ASCII = {0}",entradaUsuario);
14                  }
15          }
16  }
```

Listing 98. Ejemplo.

Capítulo 11

Introducción a C#

11.1. Introducción

C# (C *Sharp*) fue presentado en el verano del 2000 en su versión 1.0, junto con el framework .NET por Microsoft. Aunque le 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 lenguaje[1]. 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 lider del proyecto de C#) no esta 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 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 adquiriendo fuerza con el tiempo y se ha constituido en una opción real de desarrollo de software desde hace un buen tiempo.

[1] Programa 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).

11.2. 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 *MonoDevelop*. 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

11.3. Fundamentos de C#

11.3.1. 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

11.3.2. 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*.

11.3.3. 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.

11.3.4. 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

11.3.5. 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) { ... }
```

11.3.6. 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 99. 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                         Console.WriteLine();
11                         Console.Write("Leyendo entrada con Read(): ");
```

```
12                          entradaUsuario = Console.Read();
13                          Console.WriteLine("Valor ASCII = {0}",entradaUsuario);
14                  }
15          }
16  }
```

Listing 100. Ejemplo.

Capítulo 12

Introducción a Scala

12.1. 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

12.2. 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, Java
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 sistema[1]:

```
$scala
Welcome to Scala version 2.7.5.final (Java HotSpot(TM) Client VM, Java 1.5.0_19).
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:

```
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:

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

```
>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.

12.3. "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 101. 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 102. Ejemplo 2 de "Hola, Mundo.[en] Scala.

12.4. Fundamentos de Scala

12.4.1. Convenciones léxicas

Scala utiliza la misma convección 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 de paréntesis en los números:

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

ya que 1. es una coincidencia valida 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.

[2] Ver: 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 12.1. Literales en Scala

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 12.2. Secuencias de escape en Scala

12.4.2. 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 **??**

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
```

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:

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

Las literales funciones se pueden escribir de dos formas:

```
val f1: (Int,String) => = (i,s) => s+i
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:

```
val t1: (Int,String) = (1,"Dos")
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:

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

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

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

O se puede construir una tupla a partir de otra:

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

Otra forma de definir una tupla es con $->$, por ejemplo (1,Üno") es lo mismo que $1->$"*Uno*".

12.4.3. 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*:

```
scala> val array: Array[String] = new Array(5)
   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 marcara 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:

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

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

```
scala> var variable: Double = 100.0
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.

12.4.4. 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.

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

12.4.5. 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 par 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 }
```

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 13

Introducción a Scala

13.1. 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

13.2. 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, Java
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 sistema[1]:

```
$scala
Welcome to Scala version 2.7.5.final (Java HotSpot(TM) Client VM, Java 1.5.0_19).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Como cualquier intérprete, es posible empezar ejecutar expresiones directamente:

```
scala> 5+10
res0: Int = 15

scala> res0 * 2
res1: Int = 30

scala> val hola= "Hola Mundo!"
hola: java.lang.String = Hola Mundo!

scala> print ("Hola Mundo!")
Hola Mundo!
```

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

```
scala> val fecha= new Date
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:

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

```
>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.

13.3. "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 103. 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 104. Ejemplo 2 de "Hola, Mundo.[en] Scala.

13.4. Fundamentos de Scala

13.4.1. Convenciones léxicas

Scala utiliza la misma convección 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 de paréntesis en los números:

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

ya que 1. es una coincidencia valida 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.

[2]Ver: 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 13.1. Literales en Scala

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 13.2. Secuencias de escape en Scala

13.4.2. 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 **??**

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
```

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:

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

Las literales funciones se pueden escribir de dos formas:

```
val f1: (Int,String) => = (i,s) => s+i
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:

```
val t1: (Int,String) = (1,"Dos")
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:

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

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

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

O se puede construir una tupla a partir de otra:

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

Otra forma de definir una tupla es con $->$, por ejemplo (1,Üno") es lo mismo que $1->$"*Uno*".

13.4.3. 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*:

```
scala> val array: Array[String] = new Array(5)
   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 marcara 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:

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

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

```
scala> var variable: Double = 100.0
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.

13.4.4. 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.

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

13.4.5. 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 par 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 }
```

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 14

Introducción a D

14.1. 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 reingenieria 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 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.

[1] http://forum.dlang.org/thread/l37h5s2$gd81@digitalmars.com

- **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.

14.2. 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).

14.3. Fundamentos de D

14.3.1. 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 105. 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 linea de texto. El único módulo que 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 linea 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 identificador 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.

14.3.2. 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.

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

- 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         writefln("Hello\tWorld%c\n",'\x21');
4         writefln("Have a good day%c",'\x21');
5         return 0;
6 }
```

Listing 106. 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

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

- 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 elije 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.

-

14.3.3. Variables

Una variables 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:

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 vacio: Una función que no acepta parámetros, vació, con ningún parámetro.

Definición de variables en D:

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

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Las variables también pueden ser inicializadas desde su declaración:

```
exter int d = 3, f = 5;
int d = 3, f = 5;
byte z = 22;
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:

```
import std.stdio;
        int a = 10, b =10;
        int c;
        float f;
        int main (){
                writeln("Value of a is : ", a);
                /* variable re definition: * /
                int a, b;
                int c;
                float f;
                /* Initialization * /
```

Tipo	Operadores
Operadores Aritméticos:	+, -, *, /, %, ++, –
Operadores Relacionales:	==, !=, >, <, >=, <=
Operadores Lógicos:	&&, ——, !
Operadores Bit a bit:	&, — y ^
Operadores de Asignación:	=, +=, -+, *=, /=, %=, <<=, >>=, &=, ^=, —=
Operadores Misc:	sizeof, &(direccion de una variable), * (apuntador), ? :

```
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;
18              writeln("Value of f is : ", f);
19              return 0;
20      }
```

Listing 107. 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
```

14.3.4. Operadores

Un operador es un símbolo que llama al compilador para ejecutar una operación especifica, lógica o matemática. D provee los siguientes tipos de operadores:

Precedencia de operadores

Ver Cuadro **??**

14.3.5. Arreglos

Los arreglos en D pueden ser estáticos o dinámicos.
Un arreglo estático se declara de la siguiente forma:

Sintaxis
`<tipo> [<tamaño] <variable>`

Cuadro 14.1. Precedencia de operadores en D

Categoría	Operador
Postfijo	[] ->. ++ - -
Unario	+-!∼++__ type*&sizeof
Multiplicativo	* /%
Aditivo	+ -
Shift	<<>>
Relacional	<<= >>=
Igualdad	== !=
Lógicos	AND &&
Lógicos	OR ——
Condicional	?:
Asignación	= += -= *= /=%=>>= <<= &= ^= —=
Coma	,

De tal forma que un arreglo de 10 enteros quedaría de la siguiente forma:

```
int [10] arr
```

Un arreglo dinámico pos su parte de 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);
```

```
writeln(b2);
writeln(b3);
writeln(b4);

```

La longitud de un arreglo dinámico se puede hacer asignando un nuevo valor al atributo *length*:

```
int [] a;

a.length=10;

```

14.3.6. Estructuras de control

if: una sentencia consiste de una expresión booleana seguida de una o mas declaraciones.

Sintaxis

```
if (<expresión_boleana>){
     /* se ejecuta si la expresión boleana es verdadera */
}
```

Ejemplo:

```
import std.stdio;
      int main (){
            /* Definición de variable local */
          int a = 10;

          if( a < 20 ){
                writefln("a es menor que 20" );
             }
             writefln("valor de a es : %d", a);
          return 0;
      }
```

Listing 108. 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_boleana>){

} else {

}
```

Ejemplo:

```
1   import std.stdio;
2   int main (){
3       int a = 100;
4
5       if( a < 20 ){
6           writefln("a es menor que 20" );
7       }
8       else{
9           writefln("a no es menor que 20" );
10      }
11      writefln("el valor de a es : %d", a);
12      return 0;
13  }
```

Listing 109. 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        writefln("Excelente!" );
9        break;
10    case 'B' :
11    case 'C' :
12       writefln("Bien hecho" );
13       break;
14    case 'D' :
15       writefln("Pasaste" );
16       break;
17    case 'F' :
18       writefln("Trata de nuevo" );
19       break;
20    default :
21       writefln("Reprobado" );
22    }
23    writefln("Calificación  %c", grade );
24
25    return 0;
```

```
26 }
```

Listing 110. Ejemplo de estructura *switch* en D.

while: repite una(s) instruccion(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                  writefln("Valor de a: %d", a);
7                  a++;
8          }
9          return 0;
10 }
11
```

Listing 111. 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               writefln("Valor de a: %d", a);
5           }
6           return 0;
7       }
8
```

Listing 112. 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          writefln("Valor de a: %d", a);
6          a = a + 1;
7      }while( a < 20 );
8      return 0;
9  }
```

Listing 113. 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                       writefln("valor de a: %d", a);
6                       a++;
7                       if( a > 15){
8                               break;
9                       }
10              }
11              return 0;
12      }
```

Listing 114. 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       writefln("valor de a: %d", a);
10      a++;
11      }while( a < 20 );
12      return 0;
13  }
```

Listing 115. Ejemplo de *continue* en D.

14.3.7. 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 116. Ejemplo de enumeración en D.

Propiedades:

- *init*: inicializa el primer valor en la enumeración.
- *min*: regresa el valor más pequeño de la enumerción.
- *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 m ain(string[] args){
5         writefln("Min : %d", Days.m in);
6         writefln("Max : %d", Days.m ax);
7         writefln("Size of: %d", Days.sizeof);
8         return 0;
9 }
```

Listing 117. 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 m ain(string[] args){
5         writefln("Sunday : %d", sun);
6         writefln("Monday : %d", m on);
7         return 0;
8 }
```

Listing 118. Ejemplo de enumeración anónima en D.

14.3.8. 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 compañia.empleado;
2
3 class Empleado {
4 }
5
6 module compañia.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*.

14.3.9. 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");
```

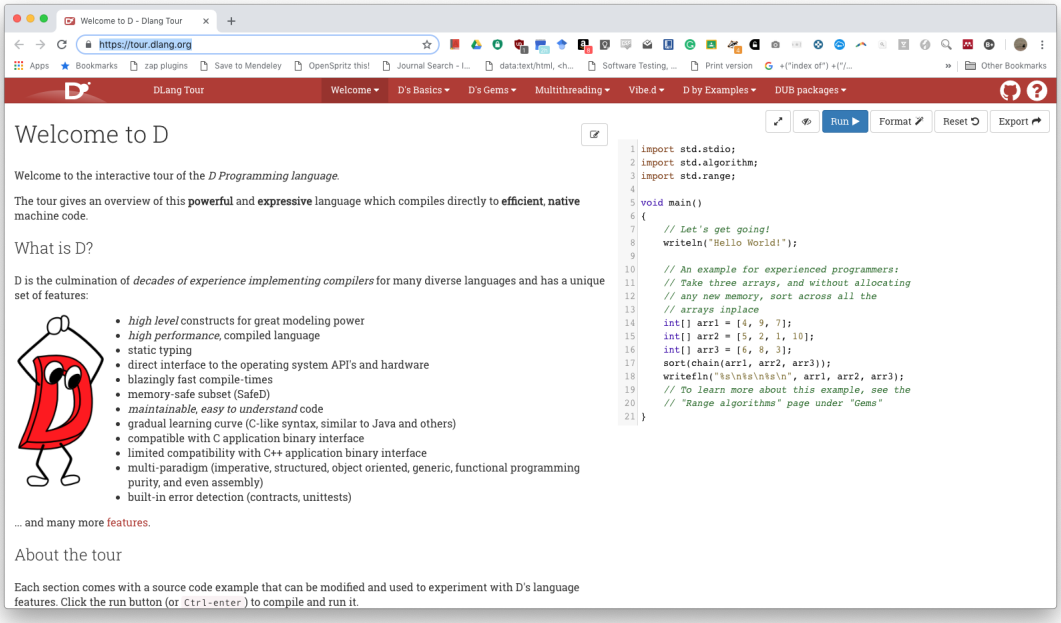
La lectura por otro lado podemos hacerla con la función *readln*() que regresa la entrada estándar a un arreglo de caracteres.

```
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*.

14.3.10. Prueba D

Una especie de tutorial interactivo puede encontrarse en : https://tour.dlang.org/



Parte II

Programación Orientada a Objetos

Capítulo 15

Introducción a D

15.1. 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 reingenieria 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 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.

[1] http://forum.dlang.org/thread/l37h5s2gd81@digitalmars.com

- **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.

15.2. 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).

15.3. Fundamentos de D

15.3.1. 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 119. 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 linea de texto. El único módulo que 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 linea 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 identificador 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.

15.3.2. 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.

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

- 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         writefln("Hello\tWorld%c\n",'\x21');
4         writefln("Have a good day%c",'\x21');
5         return 0;
6 }
```

Listing 120. 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

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

- 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 elije 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.

-

15.3.3. Variables

Una variables 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:

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 vacio: Una función que no acepta parámetros, vació, con ningún parámetro.

Definición de variables en D:

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

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Las variables también pueden ser inicializadas desde su declaración:

```
exter int d = 3, f = 5;
int d = 3, f = 5;
byte z = 22;
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:

```
import std.stdio;
        int a = 10, b =10;
        int c;
        float f;
        int main (){
                writeln("Value of a is : ", a);
                /* variable re definition: * /
                int a, b;
                int c;
                float f;
                /* Initialization * /
```

Tipo	Operadores
Operadores Aritméticos:	+, -, *, /, %, ++, –
Operadores Relacionales:	==, !=, >, <, >=, <=
Operadores Lógicos:	&&, ——, !
Operadores Bit a bit:	&, — y ^
Operadores de Asignación:	=, +=, -+, *=, /=, %=, <<=, >>=, &=, ^=, —=
Operadores Misc:	sizeof, &(direccion de una variable), * (apuntador), ? :

```
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;
18              writeln("Value of f is : ", f);
19              return 0;
20      }
```

Listing 121. 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
```

15.3.4. Operadores

Un operador es un símbolo que llama al compilador para ejecutar una operación especifica, lógica o matemática. D provee los siguientes tipos de operadores:

Precedencia de operadores

Ver Cuadro **??**

15.3.5. Arreglos

Los arreglos en D pueden ser estáticos o dinámicos.
Un arreglo estático se declara de la siguiente forma:

Sintaxis
`<tipo> [<tamaño] <variable>`

Cuadro 15.1. Precedencia de operadores en D

Categoría	Operador
Postfijo	[] ->. ++ - -
Unario	+-!∼++__ type*&sizeof
Multiplicativo	* /%
Aditivo	+ -
Shift	<<>>
Relacional	<<= >>=
Igualdad	== !=
Lógicos	AND &&
Lógicos	OR ——
Condicional	?:
Asignación	= += -= *= /=%=>>= <<= &= ^= —=
Coma	,

De tal forma que un arreglo de 10 enteros quedaría de la siguiente forma:

```
int [10] arr
```

Un arreglo dinámico pos su parte de 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);
```

```
writeln(b2);
writeln(b3);
writeln(b4);

```

La longitud de un arreglo dinámico se puede hacer asignando un nuevo valor al atributo *length*:

```
int [] a;

a.length=10;

```

15.3.6. Estructuras de control

if: una sentencia consiste de una expresión booleana seguida de una o mas declaraciones.

Sintaxis

```
if (<expresión_boleana>){
     /* se ejecuta si la expresión boleana es verdadera */
}
```

Ejemplo:

```
import std.stdio;
      int main (){
            /* Definición de variable local */
          int a = 10;

          if( a < 20 ){
                writefln("a es menor que 20" );
             }
             writefln("valor de a es : %d", a);
          return 0;
      }
```

Listing 122. 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_boleana>){

} else {

}
```

Ejemplo:

```
1   import std.stdio;
2   int main (){
3       int a = 100;
4
5       if( a < 20 ){
6           writefln("a es menor que 20" );
7       }
8       else{
9           writefln("a no es menor que 20" );
10      }
11      writefln("el valor de a es : %d", a);
12      return 0;
13  }
```

Listing 123. 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        writefln("Excelente!" );
9        break;
10    case 'B' :
11    case 'C' :
12       writefln("Bien hecho" );
13       break;
14    case 'D' :
15       writefln("Pasaste" );
16       break;
17    case 'F' :
18       writefln("Trata de nuevo" );
19       break;
20    default :
21       writefln("Reprobado" );
22    }
23    writefln("Calificación  %c", grade );
24
25    return 0;
```

```
26 }
```

Listing 124. Ejemplo de estructura *switch* en D.

while: repite una(s) instruccion(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                  writefln("Valor de a: %d", a);
7                  a++;
8          }
9          return 0;
10 }
11
```

Listing 125. 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               writefln("Valor de a: %d", a);
5           }
6           return 0;
7       }
8
```

Listing 126. 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          writefln("Valor de a: %d", a);
6          a = a + 1;
7      }while( a < 20 );
8      return 0;
9  }
```

Listing 127. 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                       writefln("valor de a: %d", a);
6                       a++;
7                       if( a > 15){
8                               break;
9                       }
10              }
11              return 0;
12      }
```

Listing 128. 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       writefln("valor de a: %d", a);
10      a++;
11      }while( a < 20 );
12      return 0;
13  }
```

Listing 129. Ejemplo de *continue* en D.

15.3.7. 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 130. Ejemplo de enumeración en D.

Propiedades:

- *init*: inicializa el primer valor en la enumeración.
- *min*: regresa el valor más pequeño de la enumerción.
- *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 m ain(string[] args){
5         writefln("Min : %d", Days.m in);
6         writefln("Max : %d", Days.m ax);
7         writefln("Size of: %d", Days.sizeof);
8         return 0;
9 }
```

Listing 131. 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 m ain(string[] args){
5         writefln("Sunday : %d", sun);
6         writefln("Monday : %d", m on);
7         return 0;
8 }
```

Listing 132. Ejemplo de enumeración anónima en D.

15.3.8. 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 compañia.empleado;
2
3 class Empleado {
4 }
5
6 module compañia.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*.

15.3.9. 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");
```

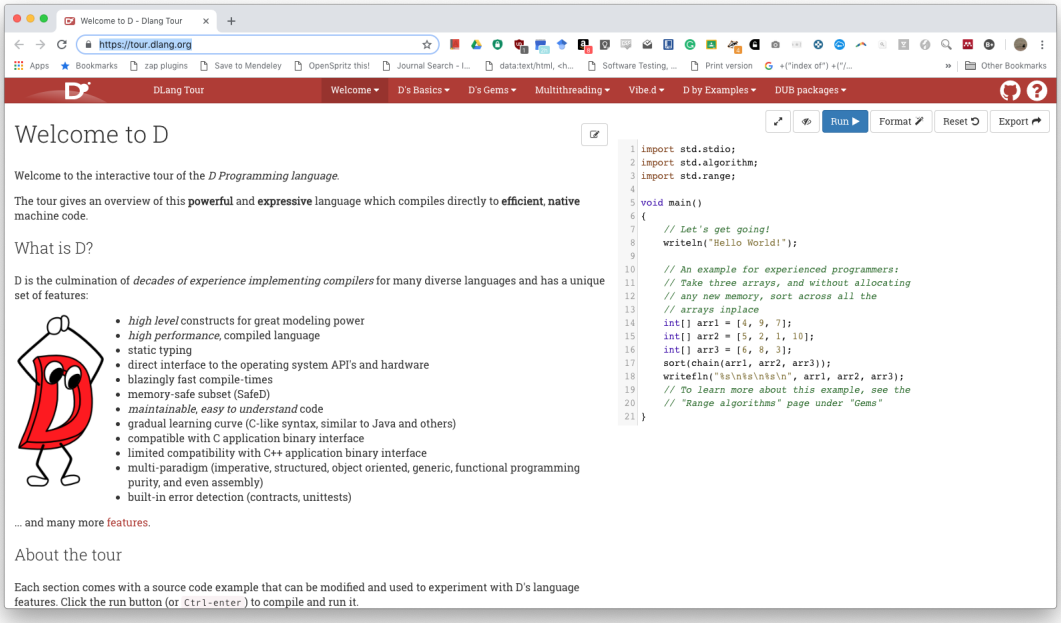
La lectura por otro lado podemos hacerla con la función *readln*() que regresa la entrada estándar a un arreglo de caracteres.

```
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*.

15.3.10. Prueba D

Una especie de tutorial interactivo puede encontrarse en : https://tour.dlang.org/



Parte III

Programación Orientada a Objetos

Capítulo 16

Introducción a la programación orientada a objetos [? ?]

16.1. 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.

16.2. 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.

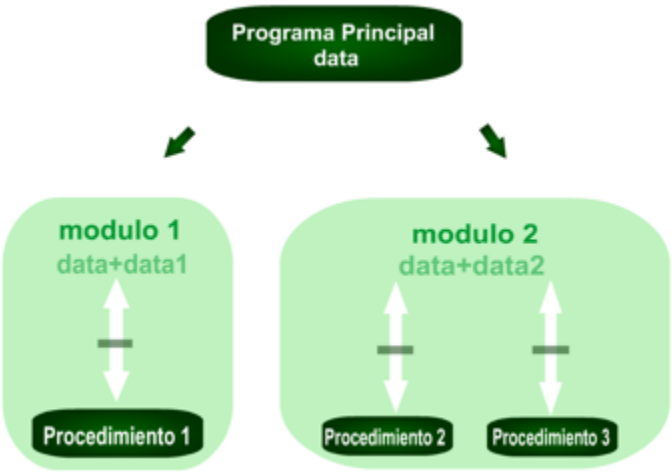
16.3. Programación modular

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.



16.4. 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.

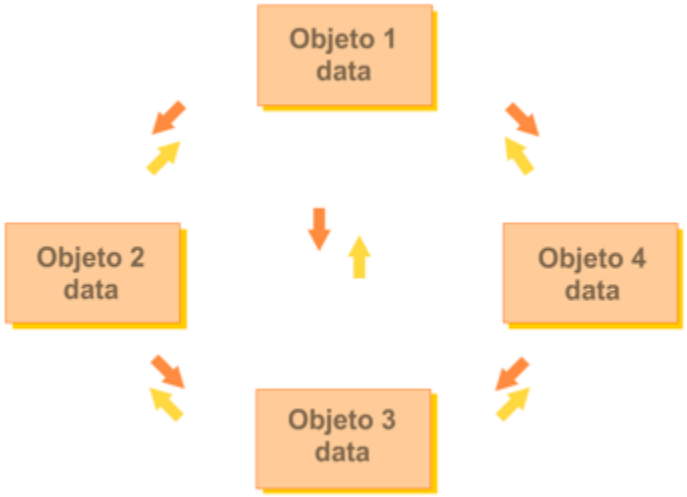
En la programación orientada a objetos, la estructura se organiza por los datos. Se escogen las representaciones de datos que mejor se ajusten a tus requerimientos. Por consecuencia, los programas se estructuran por los datos más que por las operaciones.

Los datos especifican las operaciones válidas. Ahora, los módulos agrupan representaciones de datos en forma conjunta.

16.5. 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.

16.6. 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.

16.6.1. 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.

16.6.2. 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.

16.7. 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.

16.8. 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.

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.

Cuadro 16.1. 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.

16.8.1. Características de los algunos LPOO[1]

En el Cuadro **??** podemos ver algunas características de lenguajes de programación orientados a objetos.

[1]Lenguajes de Programación Orientada a Objetos

Capítulo 17

Abstracción de datos: Clases y objetos

17.1. 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,
- ...

17.2. 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 [**?**].

17.2.1. 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.

17.3. 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 }
```

Capítulo 18

Polimorfismo AdHoc: Sobrecarga de operaciones

18.1. 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 sobrecarga[1] 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.

[1] También conocida como homonimia.

Capítulo 19

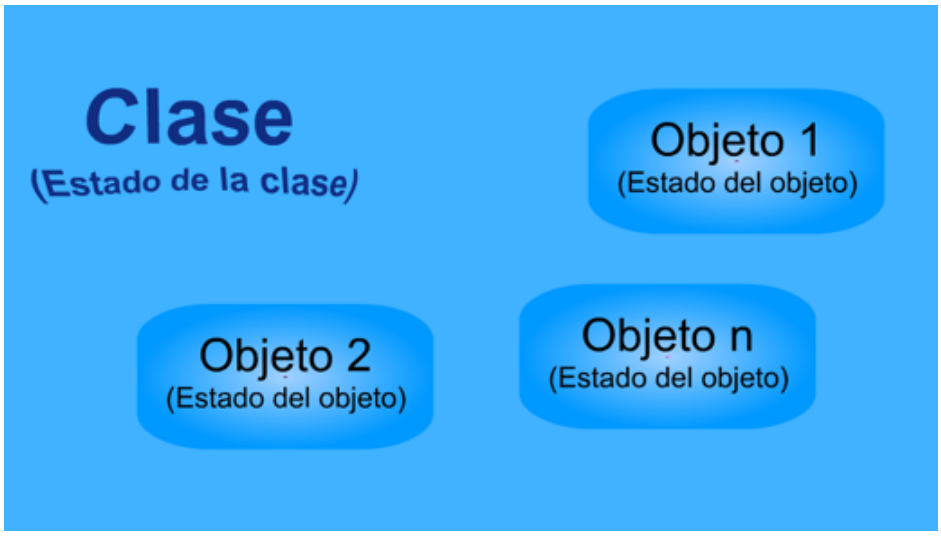
Constructores y destructores

Capítulo 20

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.



Capítulo 21

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.

Capítulo 22

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
"... la amistad corrompe el ocultamiento de información y debilita el valor del enfoque de diseño orientado a objetos"[**?**]

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 clase[1].

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.

[1]También 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 esta implementado en otros lenguajes aunque el nivel protegido permite un cierto nivel de acceso miembros de clases del mismo módulo an 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 133. 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 134. Ejemplo 2 de funciones amigas en C++.

Capítulo 23

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
"... la amistad corrompe el ocultamiento de información y debilita el valor del enfoque de diseño orientado a objetos"[**?**]

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 clase[1].

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.

[1]También 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 esta implementado en otros lenguajes aunque el nivel protegido permite un cierto nivel de acceso miembros de clases del mismo módulo an 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 135. 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 136. Ejemplo 2 de funciones amigas en C++.

Capítulo 24

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, Scala[1], C#[2] y D[3] permiten la sobrecarga de 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.

Ejercicio
Crear un programa con una clase *Pila* y los métodos correspondientes (*push, pop, getTope, getElemTope, estaVacia, estaLlena, ...*) ocupando los temas OO antes vistos de acuerdo a lo que cada lenguaje permita (constructores, miembros estáticos, sobrecarga de operaciones, sobrecarga de operadores, funciones amigas)

[1] Scala: Overloading Operators
[2] Operator Overloading Tutorial
[3] Operator Overloading

Capítulo 25

Herencia

25.1. 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

Capítulo 26

Asociaciones entre clases

26.1. 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 relaciones[1] 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).

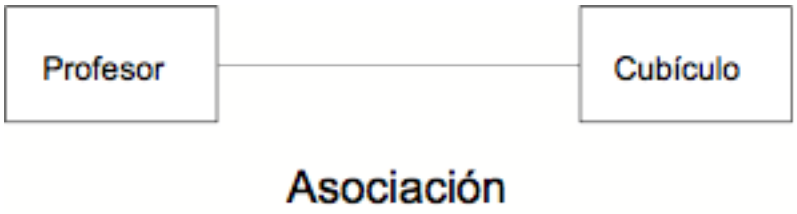


Fig. 26.1. Ejemplo de asociación en UML

[1] El 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.

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.

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 *NULL* 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.

26.2. 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**.

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.

Asociación reflexiva

Fig. 26.2. Asociación reflexiva

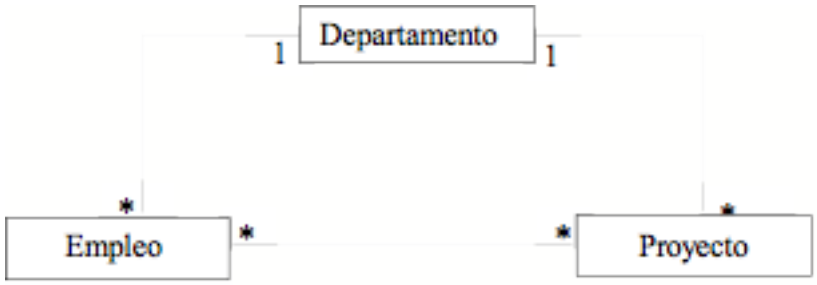


Fig. 26.3. Ejemplo de multiplicidad

26.3. 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.

26.3.1. 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.

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 27

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 *"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."*[**?**]

Ejemplo: Un automóvil se puede considerar ensamblado o agregado, donde el motor y la carrocería serian sus componentes.

El concepto de agregación puede ser relativo a la conceptualización que se tenga de los objetos que se quieran 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.

Capítulo 28

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 inclusión**[1]**[?] .** 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 29

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 redefinir 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.

Capítulo 30

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**.

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 ImprimirTodosLosElementos()* que imprime todos los elementos en el almacenamiento.
6. **Método Genérico**: Implementa un método genérico *U ProcesarDatos¡U¿(Func¡T,*

Capítulo 31

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.

Parte IV

Más allá de los Objetos

Capítulo 32

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 definir[1]:

- 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.

[1] Java assert

Capítulo 33

Diseño por contratos

33.1. Diseño por contratos en D

Capítulo 34

Pruebas de unidad

Capítulo 35

Pruebas de unidad

Capítulo 36

Multihilos: Introducción a la Programación Concurrente en Java

36.1. 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 java[1].

36.2. Concurrencia

Pero retrocedamos un poco y vamos a ponernos de acuerdo en el concepto de concurrencia. Veamos la diferencia entre concurrencia y paralelismo.

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

[1] En realidad puede existir alguna diferencia en el comportamiento de los programas, pero se ahondará en el tema más adelante.

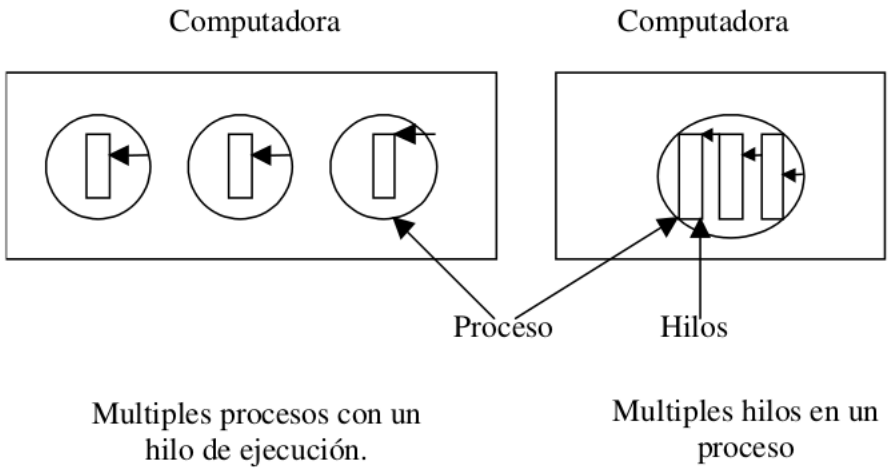


Fig. 36.1. Múltiples procesos vs. múltiples hilos en un proceso

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**.

36.3. 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 hilos[3], 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 miniprocesos, pues cada hilo se ejecuta de forma secuencial, con su propio contador de programa y pila de control. Además, al igual que los

[2]Independientemente de que algunos sistemas operativos pueden ser usados en máquinas con múltiples procesadores.

[3]También llamados **contextos de ejecución**

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 globales[4] 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.

36.4. 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*.

36.4.1. 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.

36.4.2. 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.

[4]El 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.

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 subtarea.

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:

```
class MiHilo extends Thread {

  public MiHilo (String nombre) {
    super (nombre);
  }

  public void run() {
    for( int i=0; i<4; i++){
        System.out.println( getName() + " " + i );
        try {
            sleep(400);
        } catch( InterruptedException e) { }
    }
  }
}

public class MultiHilo {
    public static void main(String arrg[]) {
        MiHilo mascar = new MiHilo("Mascando");
        MiHilo silbar = new MiHilo("Silbar");
        mascar.start();
        silbar.start();
    }
}
```

Listing 137. 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.

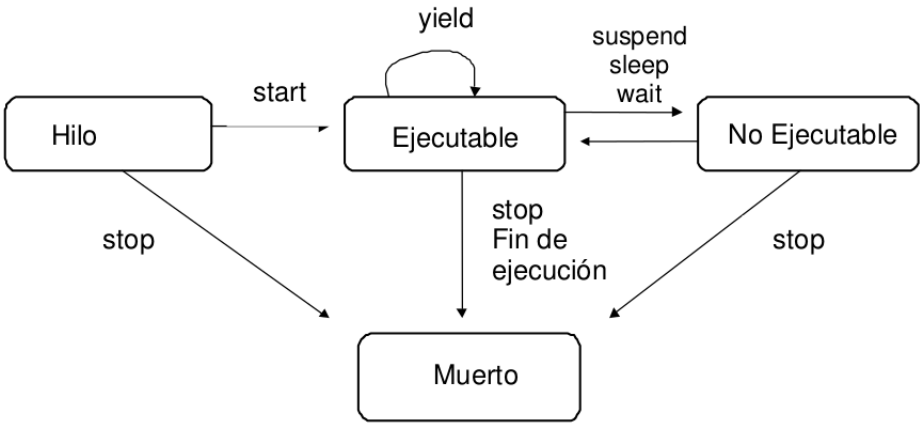


Fig. 36.2. Estados de un hilo en Java

36.5. 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 esta ”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 desaprobados en la versión 2 de Java debido a que se consideran potencialmente peligrosos para la ejecución de los programas concurrentes[5].

Veamos ahora un diagrama (figura **??**) que muestra de manera más completa los estados en los que puede estar un hilo.

36.6. 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 seria:

[5]El 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.

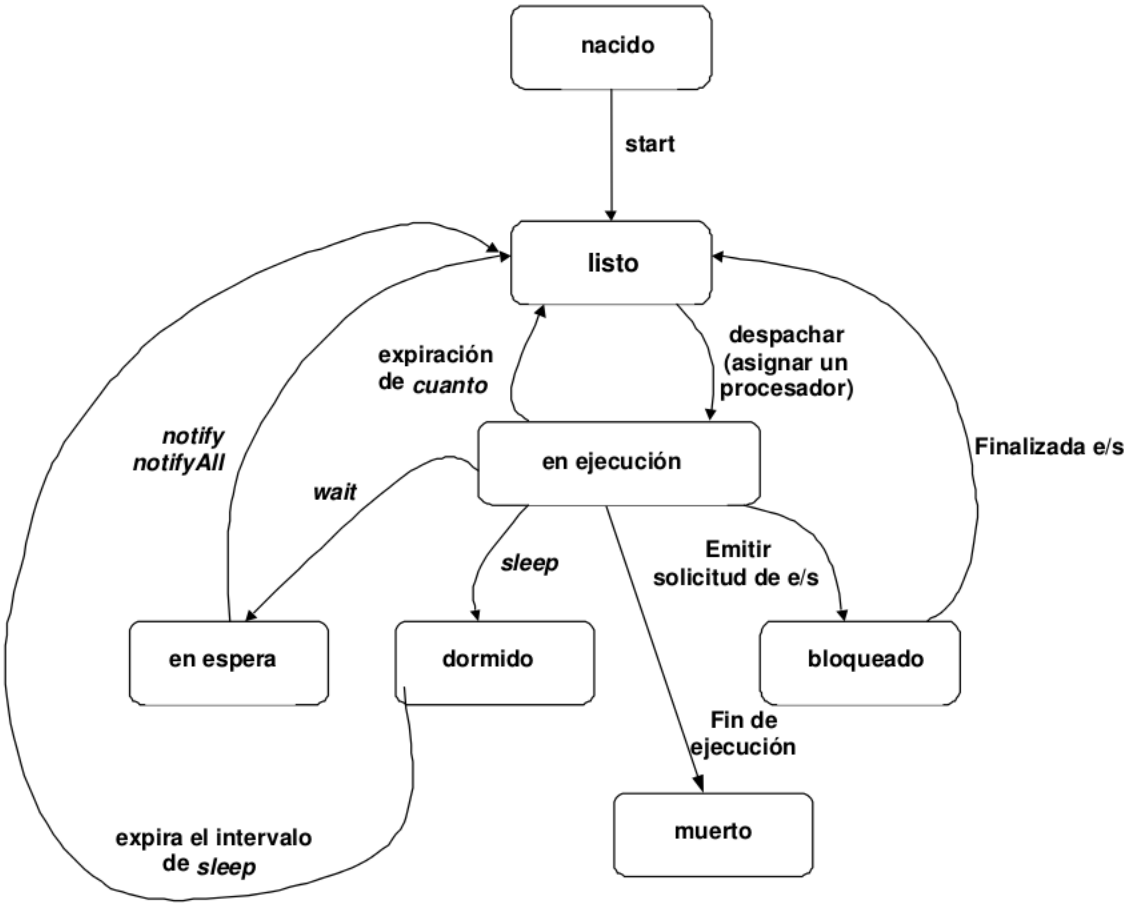


Fig. 36.3. Estados de un hilo en Java (2)

```
1 class MiHilo extends Thread {
2     public void run() {
3         . . .
4         }
5 }
```

Esta técnica, extiende a la clase *Thread*, y redefine el método *run()*, el cual debe contener un implementación propia, de acuerdo a lo que se quiera que realice el hilo.

Vamos a mencionar ahora los principales métodos de la clase[6].

- *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 redefine para controlar la ejecución del hilo.
- sleep(tiempo). Causa que el hilo se ”duermaün tiempo determinado. Un hilo dormido no compite 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 $Object$.

[6]Para las características completas ver la documentación: *Java Platform API Specification*

36.7. 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 predefinidas 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) {}
```

```
        }
    }
 }

public class prioridadHilo  {


   public static void main(String args[])
     {
       miHilo hilo_min = new miHilo ("Hilo Min");
       miHilo hilo_max = new miHilo ("Hilo Max");

       hilo_min.setPriority(Thread.MIN_PRIORITY);
       hilo_max.setPriority(Thread.MAX_PRIORITY);
       hilo_min.start();
       hilo_max.start();
     }
 }
```

Listing 138. Ejemplo de prioridades en multhilos 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:

```
// Muestra múltiples hilos desplegándose a diferentes
// intervalos.

public class PrintTest {
   public static void main( String args[] )
   {
      PrintThread thread1, thread2, thread3, thread4;

      thread1 = new PrintThread( "1" );
      thread2 = new PrintThread( "2" );
      thread3 = new PrintThread( "3" );
      thread4 = new PrintThread( "4" );

      thread1.start();
      thread2.start();
      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 139. Ejemplo múltiples hilos desplegándose a diferentes // intervalos.

36.8. 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 técnica[7].

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
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 }
```

[7] Sin embargo debería siempre considerarse el uso de *yield()* si se piensa en sistemas multiplataformas.

28

Listing 140. Ejemplo que implementa un hilo y cede el control a otros hilos.

Capítulo 37

Multihilos: Introducción a la Programación Concurrente en Java

37.1. 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 java[1].

37.2. Concurrencia

Pero retrocedamos un poco y vamos a ponernos de acuerdo en el concepto de concurrencia. Veamos la diferencia entre concurrencia y paralelismo.

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

[1] En realidad puede existir alguna diferencia en el comportamiento de los programas, pero se ahondará en el tema más adelante.

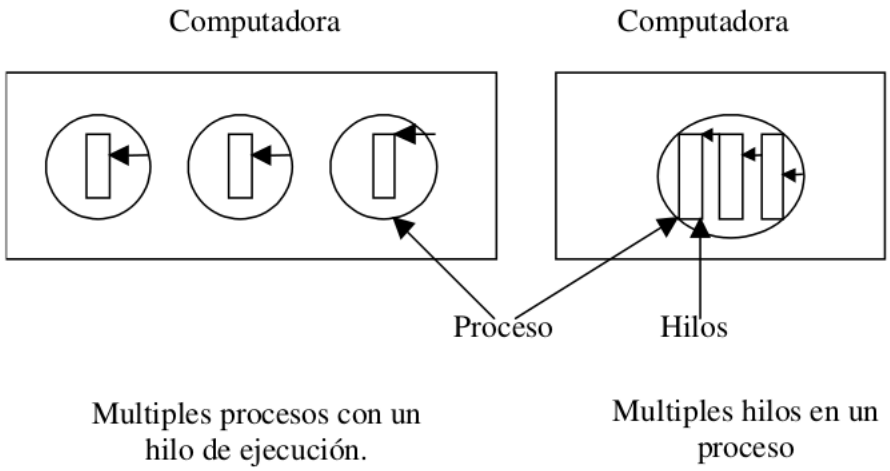


Fig. 37.1. Múltiples procesos vs. múltiples hilos en un proceso

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**.

37.3. 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 hilos[3], 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 miniprocesos, pues cada hilo se ejecuta de forma secuencial, con su propio contador de programa y pila de control. Además, al igual que los

[2]Independientemente de que algunos sistemas operativos pueden ser usados en máquinas con múltiples procesadores.
[3]También llamados **contextos de ejecución**

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 globales[4] 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.

37.4. 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*.

37.4.1. 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.

37.4.2. 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.

[4]El 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.

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 subtarea.

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 }
```

Listing 141. 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.

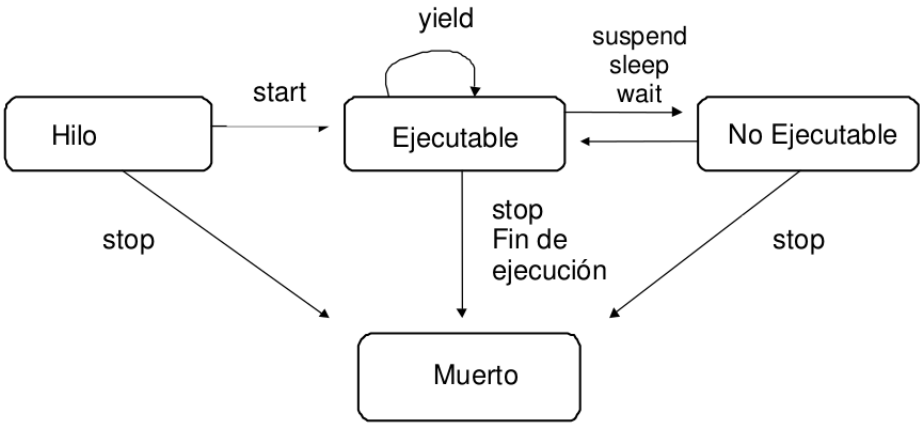


Fig. 37.2. Estados de un hilo en Java

37.5. 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 esta ”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 desaprobados en la versión 2 de Java debido a que se consideran potencialmente peligrosos para la ejecución de los programas concurrentes[5].

Veamos ahora un diagrama (figura **??**) que muestra de manera más completa los estados en los que puede estar un hilo.

37.6. 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 seria:

[5]El 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.

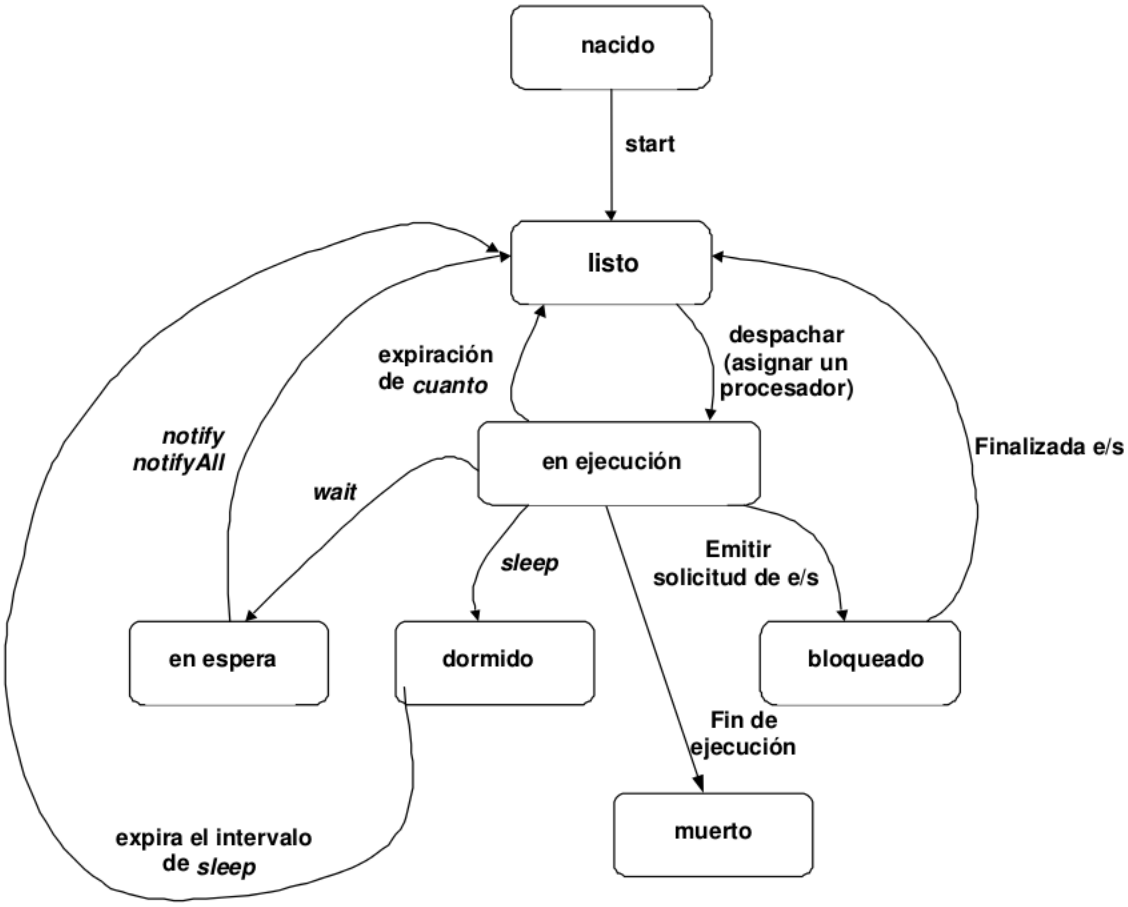


Fig. 37.3. Estados de un hilo en Java (2)

```
1 class MiHilo extends Thread {
2     public void run() {
3         . . .
4         }
5 }
```

Esta técnica, extiende a la clase *Thread*, y redefine el método *run()*, el cual debe contener un implementación propia, de acuerdo a lo que se quiera que realice el hilo.

Vamos a mencionar ahora los principales métodos de la clase[6].

- *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 redefine para controlar la ejecución del hilo.
- sleep(tiempo). Causa que el hilo se ”duermaün tiempo determinado. Un hilo dormido no compite 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 $Object$.

[6]Para las características completas ver la documentación: *Java Platform API Specification*

37.7. 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 predefinidas 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) {}
```

```
        }
    }
}

public class prioridadHilo  {


   public static void main(String args[])
     {
       miHilo hilo_min = new miHilo ("Hilo Min");
       miHilo hilo_max = new miHilo ("Hilo Max");

       hilo_min.setPriority(Thread.MIN_PRIORITY);
       hilo_max.setPriority(Thread.MAX_PRIORITY);
       hilo_min.start();
       hilo_max.start();
     }
}
```

Listing 142. Ejemplo de prioridades en multhilos 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:

```
// Muestra múltiples hilos desplegándose a diferentes
// intervalos.

public class PrintTest {
   public static void main( String args[] )
   {
      PrintThread thread1, thread2, thread3, thread4;

      thread1 = new PrintThread( "1" );
      thread2 = new PrintThread( "2" );
      thread3 = new PrintThread( "3" );
      thread4 = new PrintThread( "4" );

      thread1.start();
      thread2.start();
      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 143. Ejemplo múltiples hilos desplegándose a diferentes // intervalos.

37.8. 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 técnica[7].

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
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 }
```

[7] Sin embargo debería siempre considerarse el uso de *yield()* si se piensa en sistemas multiplataformas.

28

Listing 144. Ejemplo que implementa un hilo y cede el control a otros hilos.

Capítulo 38

Multihilos: Programación Concurrente en Java (2)

38.1. 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 sincronizado[2].

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:

```
1 class Cuenta {
2
3   static int balance = 1000;
```

[1]Un 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.

[2]Una opción adicional de implementar sincronización en Java es mediante la clase *ReentrantLock*, pero no se verá en el curso. Ver: Reentrant lock Java

```
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 145. 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 146. Ejemplo de multihilos declarando un bloque de código sincronizado.

38.1.1. 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 ocurrir[3].

Ejemplo:

```
1   public class SharedCell {
2      public static void main( String args[] )
3      {
4         HoldInteger h = new HoldInteger();
5         ProduceInteger p = new ProduceInteger( h );
6         ConsumeInteger c = new ConsumeInteger( h );
7
8         p.start();
9         c.start();
10     }
```

[3]Por 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.

```
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 147. Ejemplo: Problema Productor / Consumidor. Modificacion de un objeto compartido sin sincronizacion.

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 148. 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 149. 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 su-

puesto que hay que tratar de evitarlo[4]. El abrazo mortal se puede evitar con una cuidadosa programación que promueva el uso ordenado de los recursos[5].

38.2. 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");
```

38.3. 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 basura[6] de Java. Para que un hilo definido por 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*.

38.4. 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*.

[4] Investigar sobre el problema de los filósofos tragones.
[5] Existen algunos métodos formales que salen del alcance de este curso.
[6] garbage collector

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:

```
public class MiHilo implements Runnable {

  public void run(){
     //código del hilo
  }
}
```

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*:

```
Thread miHilo= new Thread( new MiHilo() );

miHilo.start();
```

Ejemplo:

```
public class DemoThread implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println("Hilo Hijo ");

            try {
                Thread.sleep(200);
            } catch (InterruptedException ie) {
                System.out.println("Hilo hijo se ha interrumpido! " + ie);
            }
        }

        System.out.println("Hilo hijo finalizado!");
    }

    public static void main(String[] args) {
        Thread t = new Thread(new DemoThread ());

        t.start();

```

```
        for (int i = 0; i < 3; i++) {
            System.out.println("Hilo principal");
            try {
                Thread.sleep(200);
            } catch (InterruptedException ie) {
                System.out.println("Hilo hijo interrumpido! " + ie);
            }
        }
        System.out.println("Hilo principal finalizado!");
    }
}
```

Listing 150. Ejemplo de multihilos usando la interfaz Runnable.

Ejemplo:[7]

```
// Ejemplo de implementacion de la interfaz Runnable
import java.awt.*;
import java.applet.Applet;

public class RandomCharacters extends Applet
                              implements Runnable {

   String alphabet;
   TextField output1, output2, output3;
   Button button1, button2, button3;

   Thread thread1, thread2, thread3;

   boolean suspend1, suspend2, suspend3;

   public void init()
   {
      alphabet = new String( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
      output1 = new TextField( 10 );
      output1.setEditable( false );
      output2 = new TextField( 10 );
      output2.setEditable( false );
      output3 = new TextField( 10 );
      output3.setEditable( false );
```

[7] Al compilar este programa el compilador nos avisará del uso de métodos deprecated, debido a que el código incluye llamadas a suspend() y resume().

```
25          button1 = new Button( "Suspender/Continuar 1" );
26          button2 = new Button( "Suspender/Continuar 2" );
27          button3 = new Button( "Suspender/Continuar 3" );
28
29          add( output1 );
30          add( button1 );
31          add( output2 );
32          add( button2 );
33          add( output3 );
34          add( button3 );
35       }
36
37       public void start()
38       {
39          // crea hilos e inicia cada vez que se invoca start()
40          thread1 = new Thread( this, "Hilo 1" );
41          thread2 = new Thread( this, "Hilo 2" );
42          thread3 = new Thread( this, "Hilo 3" );
43
44          thread1.start();
45          thread2.start();
46          thread3.start();
47       }
48
49       public void stop()
50       {
51          // detiene los hilos cada vez que se invoca stop()
52          thread1.stop();
53          thread2.stop();
54          thread3.stop();
55       }
56
57       public boolean action( Event event, Object obj )
58       {
59          if ( event.target == button1 )
60             if ( suspend1 ) {
61                thread1.resume();
62                suspend1 = false;
63             }
64             else {
65                thread1.suspend();
66                output1.setText( "suspendido" );
```

```
67              suspend1 = true;
68           }
69        else if ( event.target == button2 )
70           if ( suspend2 ) {
71              thread2.resume();
72              suspend2 = false;
73           }
74           else {
75              thread2.suspend();
76              output2.setText( "suspendido" );
77              suspend2 = true;
78           }
79        else if ( event.target == button3 )
80           if ( suspend3 ) {
81              thread3.resume();
82              suspend3 = false;
83           }
84           else {
85              thread3.suspend();
86              output3.setText( "suspendido" );
87              suspend3 = true;
88           }
89
90        return true;
91     }
92
93     public void run(){
94        int location;
95        char display;
96        String executingThread;
97
98        while ( true ) {
99           try {
100             Thread.sleep( (int) ( Math.random() * 5000 ) );
101          }
102          catch ( InterruptedException e ) {
103             e.printStackTrace();
104          }
105
106          location = (int) ( Math.random() * 26 );
107          display = alphabet.charAt( location );
108
```

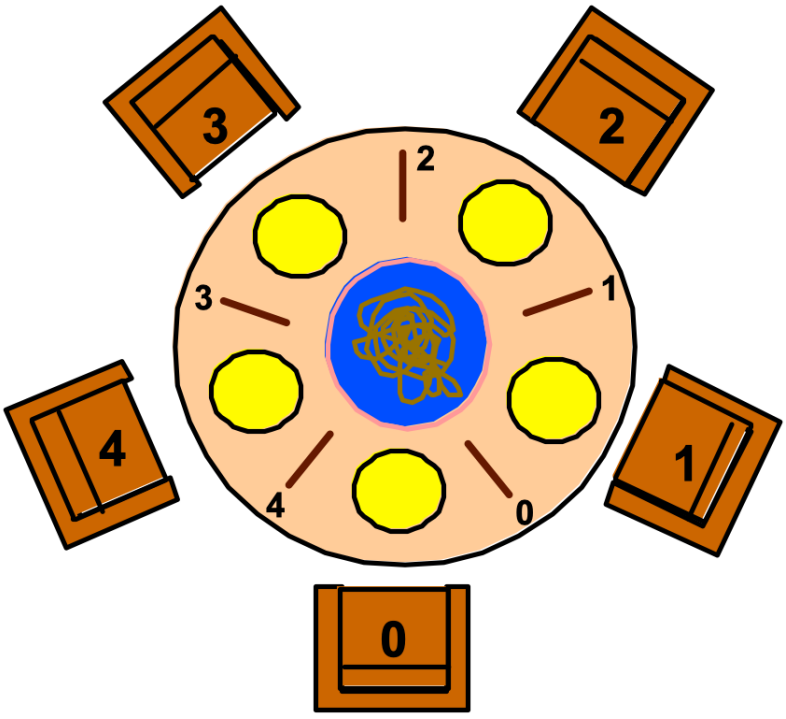


Fig. 38.1. Filósofos tragones

```
109             executingThread = Thread.currentThread().getName();
110
111             if ( executingThread.equals( "Hilo 1" ) )
112                output1.setText( "Hilo 1: " + display );
113             else if ( executingThread.equals( "Hilo 2" ) )
114                output2.setText( "Hilo 2: " + display );
115             else if ( executingThread.equals( "Hilo 3" ) )
116                output3.setText( "Hilo 3: " + display );
117          }
118       }
119    }
```

Listing 151. Ejemplo de implementacion de la interfaz *Runnable* con un *applet*.

38.5. Actividades sugeridas

Hacer un programa que simule el problema de los filósofos tragones (ver figura **??**).

Lecturas complementarias recomendadas
• Writing multithreaded Java applications [8] • Synchronization is not the enemy • Reducing contention • Threading lightly : Sometimes it's best not to share

Capítulo 39

Multihilos: Programación Concurrente en Java (2)

39.1. 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 sincronizado[2].

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:

```
1 class Cuenta {
2
3   static int balance = 1000;
```

[1]Un 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.

[2]Una opción adicional de implementar sincronización en Java es mediante la clase *ReentrantLock*, pero no se verá en el curso. Ver: Reentrant lock Java

```
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 152. 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 153. Ejemplo de multihilos declarando un bloque de código sincronizado.

39.1.1. 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 ocurrir[3].

Ejemplo:

```
1   public class SharedCell {
2      public static void main( String args[] )
3      {
4         HoldInteger h = new HoldInteger();
5         ProduceInteger p = new ProduceInteger( h );
6         ConsumeInteger c = new ConsumeInteger( h );
7
8         p.start();
9         c.start();
10     }
```

[3]Por 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.

```
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 154. Ejemplo: Problema Productor / Consumidor. Modificacion de un objeto compartido sin sincronizacion.

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 155. 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 156. 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 su-

puesto que hay que tratar de evitarlo[4]. El abrazo mortal se puede evitar con una cuidadosa programación que promueva el uso ordenado de los recursos[5].

39.2. 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");
```

39.3. 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 basura[6] de Java. Para que un hilo definido por 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*.

39.4. 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*.

[4] Investigar sobre el problema de los filósofos tragones.
[5] Existen algunos métodos formales que salen del alcance de este curso.
[6] garbage collector

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
```

```
        for (int i = 0; i < 3; i++) {
            System.out.println("Hilo principal");
            try {
                Thread.sleep(200);
            } catch (InterruptedException ie) {
                System.out.println("Hilo hijo interrumpido! " + ie);
            }
        }
        System.out.println("Hilo principal finalizado!");
    }
}
```

Listing 157. Ejemplo de multihilos usando la interfaz Runnable.

Ejemplo:[7]

```
// Ejemplo de implementacion de la interfaz Runnable
import java.awt.*;
import java.applet.Applet;

public class RandomCharacters extends Applet
                              implements Runnable {

   String alphabet;
   TextField output1, output2, output3;
   Button button1, button2, button3;

   Thread thread1, thread2, thread3;

   boolean suspend1, suspend2, suspend3;

   public void init()
   {
      alphabet = new String( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
      output1 = new TextField( 10 );
      output1.setEditable( false );
      output2 = new TextField( 10 );
      output2.setEditable( false );
      output3 = new TextField( 10 );
      output3.setEditable( false );
```

[7] Al compilar este programa el compilador nos avisará del uso de métodos deprecated, debido a que el código incluye llamadas a suspend() y resume().

```
25          button1 = new Button( "Suspender/Continuar 1" );
26          button2 = new Button( "Suspender/Continuar 2" );
27          button3 = new Button( "Suspender/Continuar 3" );
28
29          add( output1 );
30          add( button1 );
31          add( output2 );
32          add( button2 );
33          add( output3 );
34          add( button3 );
35       }
36
37       public void start()
38       {
39          // crea hilos e inicia cada vez que se invoca start()
40          thread1 = new Thread( this, "Hilo 1" );
41          thread2 = new Thread( this, "Hilo 2" );
42          thread3 = new Thread( this, "Hilo 3" );
43
44          thread1.start();
45          thread2.start();
46          thread3.start();
47       }
48
49       public void stop()
50       {
51          // detiene los hilos cada vez que se invoca stop()
52          thread1.stop();
53          thread2.stop();
54          thread3.stop();
55       }
56
57       public boolean action( Event event, Object obj )
58       {
59          if ( event.target == button1 )
60             if ( suspend1 ) {
61                thread1.resume();
62                suspend1 = false;
63             }
64             else {
65                thread1.suspend();
66                output1.setText( "suspendido" );
```

```
67              suspend1 = true;
68           }
69        else if ( event.target == button2 )
70           if ( suspend2 ) {
71              thread2.resume();
72              suspend2 = false;
73           }
74           else {
75              thread2.suspend();
76              output2.setText( "suspendido" );
77              suspend2 = true;
78           }
79        else if ( event.target == button3 )
80           if ( suspend3 ) {
81              thread3.resume();
82              suspend3 = false;
83           }
84           else {
85              thread3.suspend();
86              output3.setText( "suspendido" );
87              suspend3 = true;
88           }
89
90        return true;
91     }
92
93     public void run(){
94        int location;
95        char display;
96        String executingThread;
97
98        while ( true ) {
99           try {
100             Thread.sleep( (int) ( Math.random() * 5000 ) );
101          }
102          catch ( InterruptedException e ) {
103             e.printStackTrace();
104          }
105
106          location = (int) ( Math.random() * 26 );
107          display = alphabet.charAt( location );
108
```

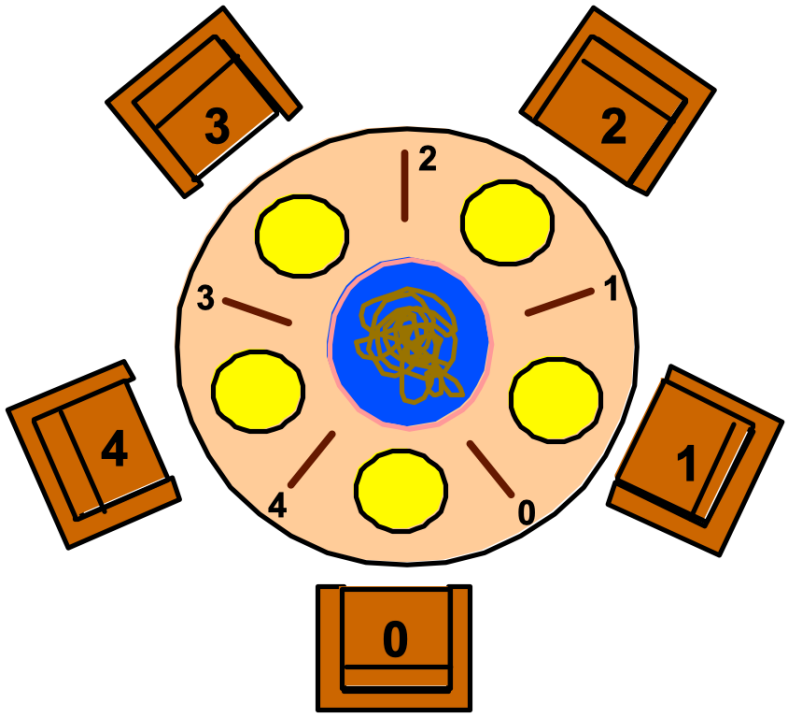


Fig. 39.1. Filósofos tragones

```
109          executingThread = Thread.currentThread().getName();
110
111          if ( executingThread.equals( "Hilo 1" ) )
112             output1.setText( "Hilo 1: " + display );
113          else if ( executingThread.equals( "Hilo 2" ) )
114             output2.setText( "Hilo 2: " + display );
115          else if ( executingThread.equals( "Hilo 3" ) )
116             output3.setText( "Hilo 3: " + display );
117       }
118    }
119 }
```

Listing 158. Ejemplo de implementacion de la interfaz *Runnable* con un *applet*.

39.5. Actividades sugeridas

Hacer un programa que simule el problema de los filósofos tragones (ver figura **??**).

Lecturas complementarias recomendadas
• Writing multithreaded Java applications [8] • Synchronization is not the enemy • Reducing contention • Threading lightly : Sometimes it's best not to share

Capítulo 40

Multihilos en Python

40.1. 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():
```

```
    while True:
        newpid = os.fork()
        if newpid == 0:
            hijo()
        else:
            print('Hola desde el proceso padre', os.getpid(), newpid)
        if input("Salir(s)") == 's':
            break

padre()
```

Listing 159. 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.

```
"""
Inicia 5 copias del programa corriendo en paralelo con el original;
cada copia cuenta hasta 5 en el mismo flujo de salida estándar,
bifurca copiando la memoria del proceso.
No trabaja en Windows sin Cygwin
"""

import os, time

def contador(count):              #ejecuta un nuevo proceso
    for i in range(count):
        time.sleep(1)            # simula que trabaja
        print('[%s] => %s' % (os.getpid(), i))

for i in range(5):
    pid = os.fork()
    if pid != 0:
        print('Proceso %d generado' % pid)  # en proceso padre: continua
    else:
        contador(5)               # sino in proceso hijo/nuevo
        os._exit(0)              # ejecuta función y sale

print('Saliendo del proceso principal.') #proceso padre no necesita esperar

```

Listing 160. Ejemplo de bifurcación de 5 procesos (Forking).

40.1.1. 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 161. 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.

40.2. 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*.

40.2.1. 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.

```
"lanza hilos hasta que se introduce 's'"

import _thread

def hijo(tid):
  print('Hola desde el hilo', tid)

def padre():
  i= 0
  while True:
    i += 1
    _thread.start_new_thread(hijo, (i,))
    if input("s para salir: ") == 's':
      break

padre()

```

Listing 162. 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.

```
import _thread

def acción(i):
  print(i ** 32)

class Potencia:
  def __init__(self, i):
    self.i = i
  def acción(self):
    print(self.i ** 32)

_thread.start_new_thread(acción, (2,))

_thread.start_new_thread((lambda: acción(2)), ())

obj = Potencia(2)
_thread.start_new_thread(obj.acción, ())
```

```
18
```

Listing 163. 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('[%s] => %s' % (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 164. 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('[%s] => %s' % (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 165. 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('[%s] => %s' % (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 166. 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('[%s] => %s' % (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 167. Ejemplo con una lista de enteros para saber que un hilo ha terminado.

40.2.2. 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 multhilo de alto nivel similar a Java con método join (sin mutexes ni variables
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('[%s] => %s' % (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 168. 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 usas 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):                # redefine run con la acción
14     print(self.i ** 32)
15
16 MiHilo(2).start()               #start() invoca run()
17
18 # pasar acción en
19 hilo = threading.Thread(target=(lambda: accion(2)))   #run() invoca target
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 169. 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 conmódulo básico de hilo
24 _thread.start_new_thread(obj.accion, ())
25 _thread.start_new_thread(accion(2), ())
26
```

Listing 170. Ejemplo de uso de módulo *threading* con referencia anidada.

40.2.3. 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 si 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(productor, (i, cola_datos))
34   time.sleep(((numproductores-1) * nummensajes) + 1)
35   print('Salida de hilo principal.')
36
```

Listing 171. Ejemplo de uso de módulo *queue* productor/consumidor.

Capítulo 41

Multihilos en Python

41.1. 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():
```

```
    while True:
        newpid = os.fork()
        if newpid == 0:
            hijo()
        else:
            print('Hola desde el proceso padre', os.getpid(), newpid)
        if input("Salir(s)") == 's':
            break

padre()
```

Listing 172. 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.

```
"""
Inicia 5 copias del programa corriendo en paralelo con el original;
cada copia cuenta hasta 5 en el mismo flujo de salida estándar,
bifurca copiando la memoria del proceso.
No trabaja en Windows sin Cygwin
"""

import os, time

def contador(count):              #ejecuta un nuevo proceso
    for i in range(count):
        time.sleep(1)            # simula que trabaja
        print('[%s] => %s' % (os.getpid(), i))

for i in range(5):
    pid = os.fork()
    if pid != 0:
        print('Proceso %d generado' % pid)  # en proceso padre: continua
    else:
        contador(5)               # sino in proceso hijo/nuevo
        os._exit(0)              # ejecuta función y sale

print('Saliendo del proceso principal.') #proceso padre no necesita esperar

```

Listing 173. Ejemplo de bifurcación de 5 procesos (Forking).

41.1.1. 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 174. 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.

41.2. 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*.

41.2.1. 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.

```
"lanza hilos hasta que se introduce 's'"

import _thread

def hijo(tid):
  print('Hola desde el hilo', tid)

def padre():
  i= 0
  while True:
    i += 1
    _thread.start_new_thread(hijo, (i,))
    if input("s para salir: ") == 's':
      break

padre()

```

Listing 175. 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.

```
import _thread

def acción(i):
  print(i ** 32)

class Potencia:
  def __init__(self, i):
    self.i = i
  def acción(self):
    print(self.i ** 32)

_thread.start_new_thread(acción, (2,))

_thread.start_new_thread((lambda: acción(2)), ())

obj = Potencia(2)
_thread.start_new_thread(obj.acción, ())
```

```
18
```

Listing 176. 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('[%s] => %s' % (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 177. 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('[%s] => %s' % (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 178. 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('[%s] => %s' % (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 179. 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('[%s] => %s' % (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 180. Ejemplo con una lista de enteros para saber que un hilo ha terminado.

41.2.2. 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 multhilo de alto nivel similar a Java con método join (sin mutexes ni variables
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('[%s] => %s' % (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 181. 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 usas 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):                # redefine run con la acción
14     print(self.i ** 32)
15
16 MiHilo(2).start()               #start() invoca run()
17
18 # pasar acción en
19 hilo = threading.Thread(target=(lambda: accion(2)))   #run() invoca target
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 182. 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 conmódulo básico de hilo
24 _thread.start_new_thread(obj.accion, ())
25 _thread.start_new_thread(accion(2), ())
26
```

Listing 183. Ejemplo de uso de módulo *threading* con referencia anidada.

41.2.3. 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 si 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(productor, (i, cola_datos))
34   time.sleep(((numproductores-1) * nummensajes) + 1)
35   print('Salida de hilo principal.')
36
```

Listing 184. Ejemplo de uso de módulo *queue* productor/consumidor.

Capítulo 42

Java: Clases anidadas y anónimas

42.1. Clases anidadas

Las clases anidadas son clases definidas dentro del alcance de otras clases. Existen dos tipos de clases anidadas[1]:

- 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 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

[1]Fuente: Nested class

- 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<Interger>
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;
31
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 185. 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 186. Ejemplo variables con el mismo nombre en clases internas .

42.2. 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 vez[2].

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");
14              }
15              public void saludarAlguien(String alguien) {
16                  nombre = alguien;
17                  System.out.println("Hello " + nombre);
18              }
```

[2] Anonymouse classes

```
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 187. Ejemplo de clases anónimas.

Capítulo 43

Java: Clases anidadas y anónimas

43.1. Clases anidadas

Las clases anidadas son clases definidas dentro del alcance de otras clases. Existen dos tipos de clases anidadas[1]:

- 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 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

[1]Fuente: Nested class

- 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<Interger>
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;
31
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 188. 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 189. Ejemplo variables con el mismo nombre en clases internas .

43.2. 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 vez[2].

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");
14              }
15              public void saludarAlguien(String alguien) {
16                  nombre = alguien;
17                  System.out.println("Hello " + nombre);
18              }
```

[2] Anonymouse classes

```
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 190. Ejemplo de clases anónimas.

Capítulo 44

Java: Expresiones lambda λ

44.1. 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 son[1]:

- 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 $\lambda - calculus$ [**?**].

44.2. 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 relevantes[2]:

- Una expresión puede tener de cero a n parámetros.
- El tipo del parámetro puede indicarse explícitamente o puede ser inferido del contexto.

[2] Lambda expressions java tutorial

- 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 45

Java: Expresiones lambda λ

45.1. 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 son[1]:

- 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 $\lambda - calculus$ [**?**].

45.2. 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 relevantes[2]:

- Una expresión puede tener de cero a n parámetros.
- El tipo del parámetro puede indicarse explícitamente o puede ser inferido del contexto.

[2] Lambda expressions java tutorial

- 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 46

Java: Interfaz gráfica con JavaFX

46.1. 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 nativos[1]. 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 *multitouch* (para tabletas y *smartphones*), animación 2D y 3D, reproducción de audio, video, etc.[**?**]

Una aplicación JavaFX puede ejecutarse[2]:

- 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].

El framework de JavaFX esta 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:

[1] Información sobre JavaFX y Swing: Java SE client technologies
[2] Basic deployment
[3] https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm

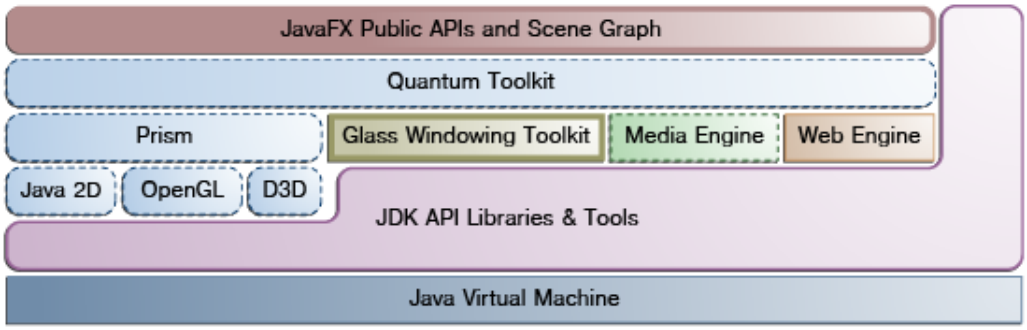


Fig. 46.1. Arquitectura de JavaFX

- *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 191. 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()1.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 192. 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 redefine 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 esce-

narios. 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 193. Ejemplo de JavaFX con múltiples escenarios.

46.2. 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;
```

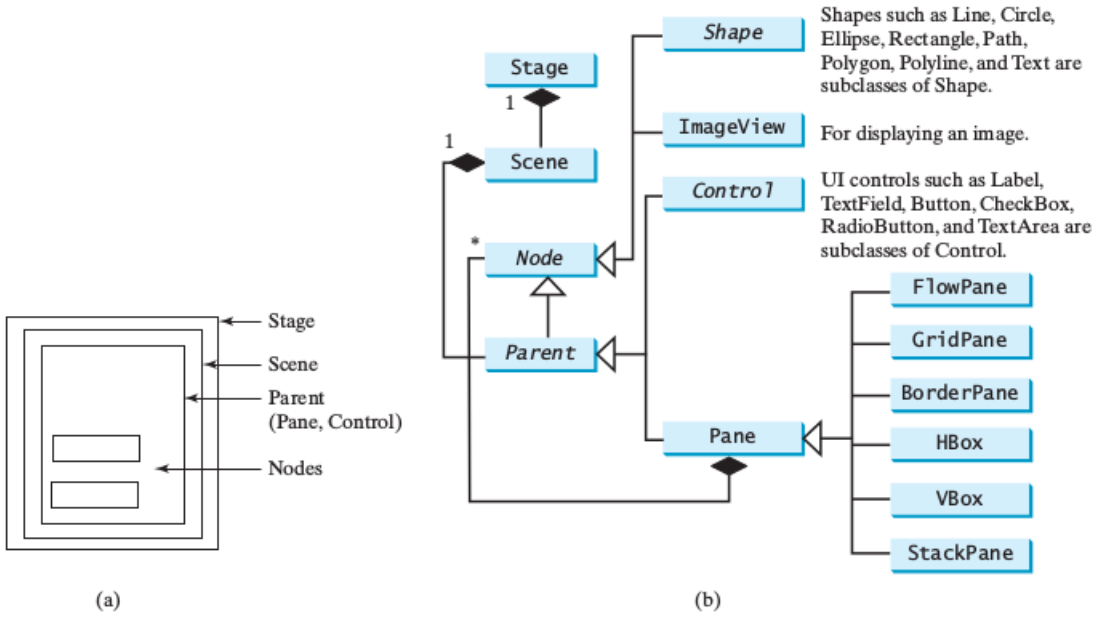


Fig. 46.2. (a) Paneles son usados para mantener nodos. (b) Nodos pueden ser figuras, vstas de imágenes, controles IU, y paneles.

```
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 194. 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.

46.2.1. Diseño de paneles

De forma similar a las otras bibliotecas de IU, JavaFX ofrece un conjunto de tipos de páneles 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 .
```

```
                //Parámetros: top - the top offset;
                //   right - the right offset;
                // bottom - the bottom offset; left - the left offset.
                //Fuente:
                // https://docs.oracle.com/javase/8/javafx/api/javafx/geometry/Insets.html
        panel.setPadding(new Insets(11, 12, 13, 14)1);
        panel.setHgap(5);
        panel.setVgap(5);
        // Coloca los nodos en el panel
        panel.getChildren().addAll(new Label("Nombre:"),
            new TextField());
        TextField tfIniciales = new TextField();
        tfIniciales.setPrefColumnCount(2);
        panel.getChildren().addAll(new Label("Apellidos:"),
            new TextField(), new Label("Iniciales:"), tfIniciales);
        // Crea una escena y la coloca en el escenario
        Scene escena = new Scene(panel, 200, 250);
        escenarioPrimario.setTitle("MuestraFlowPane");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
    }
}
```

Listing 195. Ejemplo JavaFX de panel usando *FlowPane*.

Usando BorderPane. Ejemplo:

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class MuestraBorderPane extends Application {
    @Override
    public void start(Stage escenarioPrimario) {
        // Crea un BorderPane
        BorderPane panel = new BorderPane();
        // Coloca nodos en el panel
        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 196. 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 47

Java: Interfaz gráfica con JavaFX

47.1. 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 nativos[1]. 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 *multitouch* (para tabletas y *smartphones*), animación 2D y 3D, reproducción de audio, video, etc.[**?**]

Una aplicación JavaFX puede ejecutarse[2]:

- 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].

El framework de JavaFX esta 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:

[1] Información sobre JavaFX y Swing: Java SE client technologies
[2] Basic deployment
[3] https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm

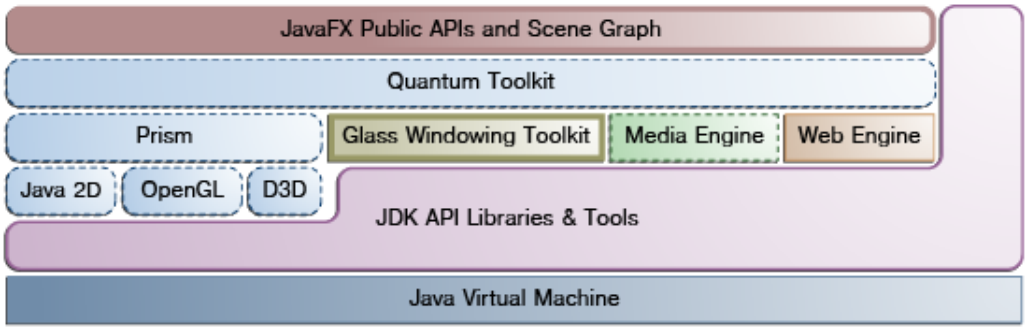


Fig. 47.1. Arquitectura de JavaFX

- *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 197. 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()1.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 198. 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 redefine 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 esce-

narios. 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 butón en el escenario
18         escenario.setScene(new Scene(new Button("Nuevo escenario"), 100, 100));
19         escenario.show();
20     }
21 }
```

Listing 199. Ejemplo de JavaFX con múltiples escenarios.

47.2. 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;
```

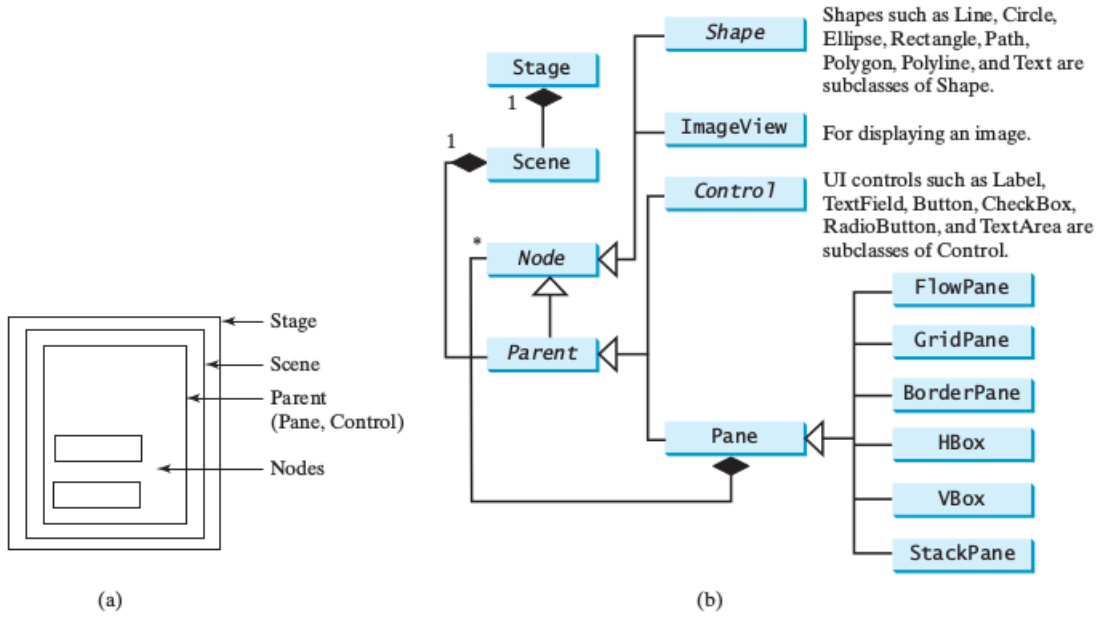


Fig. 47.2. (a) Paneles son usados para mantener nodos. (b) Nodos pueden ser figuras, vstas de imágenes, controles IU, y paneles.

```
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 200. 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.

47.2.1. Diseño de paneles

De forma similar a las otras bibliotecas de IU, JavaFX ofrece un conjunto de tipos de páneles 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 .
```

```
                //Parámetros: top - the top offset;
                //   right - the right offset;
                // bottom - the bottom offset; left - the left offset.
                //Fuente:
                // https://docs.oracle.com/javase/8/javafx/api/javafx/geometry/Insets.html
        panel.setPadding(new Insets(11, 12, 13, 14)1);
        panel.setHgap(5);
        panel.setVgap(5);
        // Coloca los nodos en el panel
        panel.getChildren().addAll(new Label("Nombre:"),
            new TextField());
        TextField tfIniciales = new TextField();
        tfIniciales.setPrefColumnCount(2);
        panel.getChildren().addAll(new Label("Apellidos:"),
            new TextField(), new Label("Iniciales:"), tfIniciales);
        // Crea una escena y la coloca en el escenario
        Scene escena = new Scene(panel, 200, 250);
        escenarioPrimario.setTitle("MuestraFlowPane");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
    }
}
```

Listing 201. Ejemplo JavaFX de panel usando *FlowPane*.

Usando BorderPane. Ejemplo:

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class MuestraBorderPane extends Application {
    @Override
    public void start(Stage escenarioPrimario) {
        // Crea un BorderPane
        BorderPane panel = new BorderPane();
        // Coloca nodos en el panel
        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 202. 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 48

Java: Manejo de eventos con JavaFX

48.1. 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 <T extends Event>*. Esta interfaz define el comportamiento común para todos los manejadores. Por su parte, *<T extends Event>* especifica que *T* es un tipo genérico que es un subtipo de *Event*.

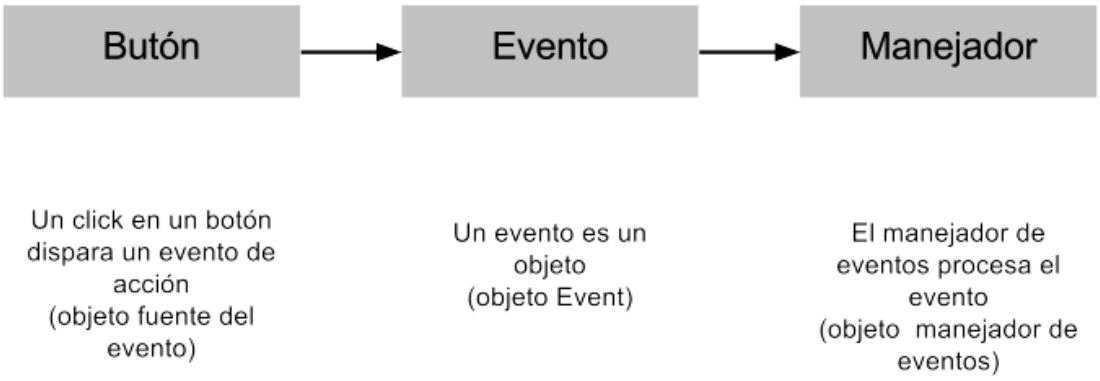


Fig. 48.1. Proceso de manejo de evento

- El objeto manejador de *EventHandler* debe estar registrado con el objeto fuente del evento con el método $fuente.setOnAction(manejador)$.

La interfaz *EventHandler<ActionEvent>* contiene el método $handle(ActionEvent)$ para procesar el evento de acción. Nuestra clase manejadora debe redefinir 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 203. Ejemplo JavaFX manejo de evento acción para 2 botones .

Como se aprecia, se declararón 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.

48.2. 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.

48.3. 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 204. 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 205. Ejemplo JavaFX dos botones con manejo de eventos.

48.3.1. 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. Ejemplo[1]:

```
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();
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              }
```

[1] Anonymous classes

```
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 206. Ejemplo Java FX manejador de eventos con clase anónimas.

Otro ejemplo, en este caso el programa usa una clase anónima donde redefine 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 207. Ejemplo JavaFX con clases anónimas y uso de *TextField*.

48.3.2. Usando expresiones lambda

Las expresiones lambda introducidad 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 butón 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 208. Ejemplo JavaFX y manejador de eventos con expresiones lambda.

48.3.3. 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);
```

```
        escenarioPrimario.setTitle("Ejemplo de Evento de Mouse");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
    }
}
```

Listing 209. Ejemplo JavaFX con evento de mouse.

48.3.4. Ejemplo con evento de teclado

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class EjemploEventoTeclado extends Application {
    @Override
    public void start(Stage escenarioPrimario) {

        Pane panel = new Pane();
        panel.setMinSize(300, 300);
        Text texto = new Text(20, 20, "A");
        panel.getChildren().add(texto);
        texto.setOnKeyPressed(e -> {
            switch (e.getCode()) {
            case DOWN: texto.setY(texto.getY() + 10); break;
            case UP: texto.setY(texto.getY() - 10); break;
            case LEFT: texto.setX(texto.getX() - 10); break;
            case RIGHT: texto.setX(texto.getX() + 10); break;
            default:
                if (Character.isLetterOrDigit(e.getText().charAt(0)))
                    texto.setText(e.getText());
            }
        });

        Scene escena = new Scene(panel);
        escenarioPrimario.setTitle("Ejemplo Evento Teclado");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
        texto.requestFocus(); // texto se enfoca para recibir la entrada de teclado
```

```
32      }
33  }
```

Listing 210. Ejemplo JavaFX con evento de teclado.

48.3.5. 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();
```

```
            } else if (e.getButton() == MouseButton.SECONDARY) {
                panelCírculo.reducir();
            }
        });

        panelCírculo.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.U) {
                panelCírculo.agrandar();
            } else if (e.getCode() == KeyCode.D) {
                panelCírculo.reducir();
            }
        });

        BorderPane panelBorde = new BorderPane();
        panelBorde.setCenter(panelCírculo);
        panelBorde.setBottom(hBox);
        BorderPane.setAlignment(hBox, Pos.CENTER);

        Scene escena = new Scene(panelBorde, 200, 150);
        escenarioPrimario.setTitle("EjemploEventoTecladoMouse");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
        panelCírculo.requestFocus();
    }
}

// vista en ejemplo anterior
class PanelCírculo extends StackPane {
    private Circle círculo = new Circle(50);

    public PanelCírculo() {
        getChildren().add(círculo);
        círculo.setStroke(Color.BLACK);
        círculo.setFill(Color.WHITE);
    }

    public void agrandar() {
        círculo.setRadius(círculo.getRadius() + 2);
    }

    public void reducir() {
        círculo.setRadius(círculo.getRadius() > 2 ?
```

```
77              círculo.getRadius() - 2 : círculo.getRadius());
78      }
79  }
```

Listing 211. Ejemplo JavaFX con evento de teclado y mouse.

48.3.6. 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 212. Ejemplo JavaFX con un objeto observable.

Capítulo 49

Java: Manejo de eventos con JavaFX

49.1. 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 <T extends Event>*. Esta interfaz define el comportamiento común para todos los manejadores. Por su parte, *<T extends Event>* especifica que *T* es un tipo genérico que es un subtipo de *Event*.

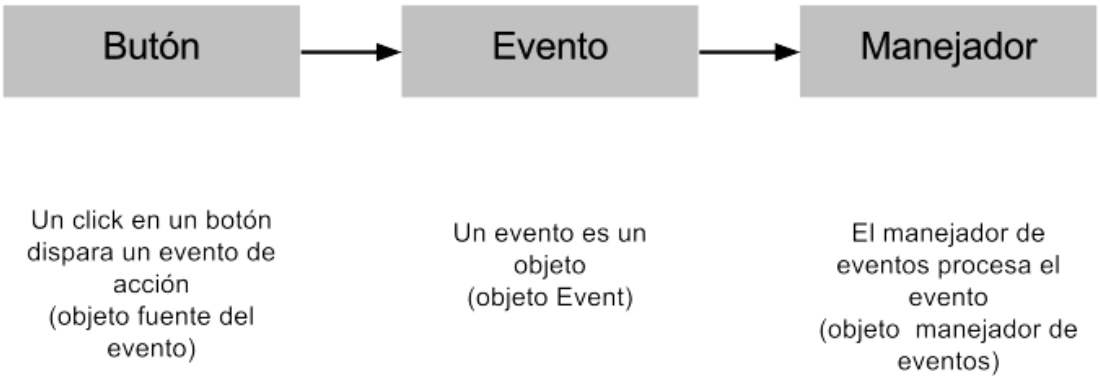


Fig. 49.1. Proceso de manejo de evento

- El objeto manejador de *EventHandler* debe estar registrado con el objeto fuente del evento con el método $fuente.setOnAction(manejador)$.

La interfaz *EventHandler<ActionEvent>* contiene el método $handle(ActionEvent)$ para procesar el evento de acción. Nuestra clase manejadora debe redefinir 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 213. Ejemplo JavaFX manejo de evento acción para 2 botones .

Como se aprecia, se declararón 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.

49.2. 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.

49.3. 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 214. 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 215. Ejemplo JavaFX dos botones con manejo de eventos.

49.3.1. 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. Ejemplo[1]:

```
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();
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              }
```

[1] Anonymous classes

```
        });

        StackPane raíz = new StackPane();
        raíz.getChildren().add(btn);
        escenarioPrimario.setScene(new Scene(raíz, 300, 250));
        escenarioPrimario.show();
    }
}
```

Listing 216. Ejemplo Java FX manejador de eventos con clase anónimas.

Otro ejemplo, en este caso el programa usa una clase anónima donde redefine la implementación de clase *TextField* para los métodos *replaceText*() y *replaceSelection*(), creando un campo de texto que solo acepta valores numéricos:

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ClaseAnónimaTextFieldAdaptado extends Application {

    final static Label etiqueta = new Label();

    @Override
    public void start(Stage escenarioPrimario) {
        Group raíz = new Group();
        Scene escena = new Scene(raíz, 300, 150);
        escenarioPrimario.setScene(escena);
        escenarioPrimario.setTitle("Ejemplo de campo de texto");

        GridPane cuadrícula = new GridPane();
        // asigna las propiedades de relleno de espacio con un objeto de Insets
        cuadrícula.setPadding(new Insets(10, 10, 10, 10));
        cuadrícula.setVgap(5);
        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 217. Ejemplo JavaFX con clases anónimas y uso de *TextField*.

49.3.2. Usando expresiones lambda

Las expresiones lambda introducidad 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 butón 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 218. Ejemplo JavaFX y manejador de eventos con expresiones lambda.

49.3.3. 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);
```

```
        escenarioPrimario.setTitle("Ejemplo de Evento de Mouse");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
    }
}
```

Listing 219. Ejemplo JavaFX con evento de mouse.

49.3.4. Ejemplo con evento de teclado

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class EjemploEventoTeclado extends Application {
    @Override
    public void start(Stage escenarioPrimario) {

        Pane panel = new Pane();
        panel.setMinSize(300, 300);
        Text texto = new Text(20, 20, "A");
        panel.getChildren().add(texto);
        texto.setOnKeyPressed(e -> {
            switch (e.getCode()) {
            case DOWN: texto.setY(texto.getY() + 10); break;
            case UP: texto.setY(texto.getY() - 10); break;
            case LEFT: texto.setX(texto.getX() - 10); break;
            case RIGHT: texto.setX(texto.getX() + 10); break;
            default:
                if (Character.isLetterOrDigit(e.getText().charAt(0)))
                    texto.setText(e.getText());
            }
        });

        Scene escena = new Scene(panel);
        escenarioPrimario.setTitle("Ejemplo Evento Teclado");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
        texto.requestFocus(); // texto se enfoca para recibir la entrada de teclado
```

```
    }
}
```

Listing 220. Ejemplo JavaFX con evento de teclado.

49.3.5. Ejemplo con evento de teclado y mouse

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.input.KeyCode;
import javafx.scene.input.MouseButton;
import javafx.scene.layout.HBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class EjemploEventoTecladoMouse extends Application {
    private PanelCírculo panelCírculo = new PanelCírculo();

    @Override
    public void start(Stage escenarioPrimario) {
        // mantener dos botones en un HBox
        HBox hBox = new HBox();
        hBox.setSpacing(10);
        hBox.setAlignment(Pos.CENTER);
        Button btAgrandar = new Button("Agrandar");
        Button btReducir = new Button("Reducir");
        hBox.getChildren().add(btAgrandar);
        hBox.getChildren().add(btReducir);

        // Crear y registrar manejadores
        btAgrandar.setOnAction(e -> panelCírculo.agrandar());
        btReducir.setOnAction(e -> panelCírculo.reducir());

        panelCírculo.setOnMouseClicked(e -> {
            if (e.getButton() == MouseButton.PRIMARY) {
                panelCírculo.agrandar();
```

```
            } else if (e.getButton() == MouseButton.SECONDARY) {
                panelCírculo.reducir();
            }
        });

        panelCírculo.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.U) {
                panelCírculo.agrandar();
            } else if (e.getCode() == KeyCode.D) {
                panelCírculo.reducir();
            }
        });

        BorderPane panelBorde = new BorderPane();
        panelBorde.setCenter(panelCírculo);
        panelBorde.setBottom(hBox);
        BorderPane.setAlignment(hBox, Pos.CENTER);

        Scene escena = new Scene(panelBorde, 200, 150);
        escenarioPrimario.setTitle("EjemploEventoTecladoMouse");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
        panelCírculo.requestFocus();
    }
}

// vista en ejemplo anterior
class PanelCírculo extends StackPane {
    private Circle círculo = new Circle(50);

    public PanelCírculo() {
        getChildren().add(círculo);
        círculo.setStroke(Color.BLACK);
        círculo.setFill(Color.WHITE);
    }

    public void agrandar() {
        círculo.setRadius(círculo.getRadius() + 2);
    }

    public void reducir() {
        círculo.setRadius(círculo.getRadius() > 2 ?
```

```
77              círculo.getRadius() - 2 : círculo.getRadius());
78      }
79  }
```

Listing 221. Ejemplo JavaFX con evento de teclado y mouse.

49.3.6. 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 222. Ejemplo JavaFX con un objeto observable.

Capítulo 50

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.

50.1. 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.

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.

Cuadro 50.1. Protocolos comunes

Acrónimo	Nombre	Descripción
HTTP	*HyperText Transport Protocol*	Protocolo de hipertexto. Es la base del World Wide Web.
FTP	*File Transfer,Protocol*	Protocolo de transferencia de archivos.
POP3	*Post Office Protocol*	Protocolo que permite el acceso al correo electrónico.
SMTP	*Simple Mail Transfer Protocol*	Protocolo para transferencia de correo electrónico.
NNTP	*Network News Transfer Protocol*	Protocolo para grupos de noticias (news)

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.

50.1.1. Clase *URL*

Esta clase permite crear instancias que almacenen direcciones de recursos en Internet[1]. 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
2  import java.applet.*;
3  import java.net.*;
4  import java.awt.*;
5
6  public class URLEjemplo extends Applet {
7
8     URL utm = null;
9
10    public void init()
11      {
12
13        try {
14            utm = new URL("http://www.utm.mx");
15          }
16        catch (MalformedURLException e)
17          {
18            System.out.println("Error:" + e.getMessage());
19          }
20      }
21
22    public boolean mouseDown(Event evt, int x, int y)
23      {
24        getAppletContext().showDocument(utm);
25        return(true);
```

[1]URL. *Uniform Resource Locator*

```
26      }
27    }
```

Listing 223. Ejemplo de clase URL.

Este programa detecta un *click* del ratón sobre el área del *applet* y como acción asociada despliega una página de html en el navegador[2].

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.

50.1.2. 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 224. 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:

[2]No se pruebe en el *appletviewer* ya que este no despliega más que el *applet* y no muestra la página html.

```
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.getByName(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 225. Ejemplo de *InetAddress*, identifica la dirección IP asociada al *host*.

Ejecutando por ejemplo:

```
\$java NSLookupApp www.utm.mx
```

50.1.3. 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.

50.1.4. 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     DataInputStream 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 DataInputStream(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 226. 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**:

```
$ java PortTalkApp yodocono.utm.mx 1234
Conectando a: yodocono.utm.mx puerto 1234.
servidor destino yodocono.utm.mx.
IP servidor destino 192.100.170.5.
numero de puerto servidor destino 1234.
servidor Local iec23.
IP servidor Local 192.100.170.33.
numero de puerto servidor Local 2162.
Enviar, recibir, o salir (E/R/S): e
Hola yodocono
Enviar, recibir, o salir (E/R/S): r
***onocodoy aloH
```

Ejemplo de ejecución del **servidor**:

```
$ java ReverServerApp
Servidor escuchando en puerto: 1234.
Aceptando conexion a iec23.utm.mx en puerto 2162.
Recibido: Hola yodocono
Enviado: onocodoy aloH
```

50.1.5. Clase *DatagramSocket*

Esta clase es el punto de entrada de todas las acciones sobre datagramas UDP[3]. Sería el equivalente a la clase *Socket* y *ServerSocket* para el protocolo TCP, ya que implementa los *sockets* cliente y servidor.

Principales métodos[4] de la clase *DatagramSocket*:

- *send()*. Enviar un datagrama a través del *socket*.
- *receive()*. Recibir un datagrama a través del *socket*.
- *close()*. Cerrar el *socket*.

50.1.6. 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
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.
```

[3]UDP 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.

[4]Algunos métodos importantes no se mencionan porque son comunes a los proporcionados por otras clases con anterioridad.

```
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: salida
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*:

```
$ java GetTimeApp

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:58 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
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.

50.2. 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("N£mero 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        outStream.writeBytes(sendLine);
90        outStream.write(13);
91        outStream.write(10);
92        outStream.flush();
93        break;
94       case 'R':
95        int inByte;
96        System.out.print("***");
97        while ((inByte = inStream.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 227. Ejemplo de uso de la clase Socket. *PortTalkApp*.

50.3. 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 conexion 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      outStream.write((byte)outLine.charAt(i));
29     outStream.write(13);
30     outStream.write(10);
31     outStream.flush();
32     System.out.println("Enviado: "+outLine);
33    } while(!finished);
34    inStream.close();
```

```
35     outStream.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 228. Ejemplo de uso de la clase *ServerSocket*.

50.4. 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 229. Ejemplo de escucha de un socket UDP. *TimeServerApp*

50.5. 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 bufferLength = 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("excepci¢n: IOException");
43   }
44  }
45 }
```

Listing 230. 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 51

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.

51.1. 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.

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.

Cuadro 51.1. Protocolos comunes

Acrónimo	Nombre	Descripción
HTTP	*HyperText Transport Protocol*	Protocolo de hipertexto. Es la base del World Wide Web.
FTP	*File Transfer,Protocol*	Protocolo de transferencia de archivos.
POP3	*Post Office Protocol*	Protocolo que permite el acceso al correo electrónico.
SMTP	*Simple Mail Transfer Protocol*	Protocolo para transferencia de correo electrónico.
NNTP	*Network News Transfer Protocol*	Protocolo para grupos de noticias (news)

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.

51.1.1. Clase *URL*

Esta clase permite crear instancias que almacenen direcciones de recursos en Internet[1]. 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
2  import java.applet.*;
3  import java.net.*;
4  import java.awt.*;
5
6  public class URLEjemplo extends Applet {
7
8     URL utm = null;
9
10    public void init()
11      {
12
13        try {
14            utm = new URL("http://www.utm.mx");
15          }
16        catch (MalformedURLException e)
17          {
18            System.out.println("Error:" + e.getMessage());
19          }
20      }
21
22    public boolean mouseDown(Event evt, int x, int y)
23      {
24        getAppletContext().showDocument(utm);
25        return(true);
```

[1]URL. *Uniform Resource Locator*

```
26      }
27    }
```

Listing 231. Ejemplo de clase URL.

Este programa detecta un *click* del ratón sobre el área del *applet* y como acción asociada despliega una página de html en el navegador[2].

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.

51.1.2. 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 232. 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:

[2]No se pruebe en el *appletviewer* ya que este no despliega más que el *applet* y no muestra la página html.

```
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.getByName(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 233. Ejemplo de *InetAddress*, identifica la dirección IP asociada al *host*.

Ejecutando por ejemplo:

```
\$java NSLookupApp www.utm.mx
```

51.1.3. 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.

51.1.4. 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     DataInputStream 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 DataInputStream(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 234. 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**:

```
$ java PortTalkApp yodocono.utm.mx 1234
Conectando a: yodocono.utm.mx puerto 1234.
servidor destino yodocono.utm.mx.
IP servidor destino 192.100.170.5.
numero de puerto servidor destino 1234.
servidor Local iec23.
IP servidor Local 192.100.170.33.
numero de puerto servidor Local 2162.
Enviar, recibir, o salir (E/R/S): e
Hola yodocono
Enviar, recibir, o salir (E/R/S): r
***onocodoy aloH
```

Ejemplo de ejecución del **servidor**:

```
$ java ReverServerApp
Servidor escuchando en puerto: 1234.
Aceptando conexion a iec23.utm.mx en puerto 2162.
Recibido: Hola yodocono
Enviado: onocodoy aloH
```

51.1.5. Clase *DatagramSocket*

Esta clase es el punto de entrada de todas las acciones sobre datagramas UDP[3]. Sería el equivalente a la clase *Socket* y *ServerSocket* para el protocolo TCP, ya que implementa los *sockets* cliente y servidor.

Principales métodos[4] de la clase *DatagramSocket*:

- *send()*. Enviar un datagrama a través del *socket*.
- *receive()*. Recibir un datagrama a través del *socket*.
- *close()*. Cerrar el *socket*.

51.1.6. 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*:

```
$ java TimeServerApp
iec23: TimeServer escuchando el puerto 2345.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:58 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
```

[3]UDP 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.

[4]Algunos métodos importantes no se mencionan porque son comunes a los proporcionados por otras clases con anterioridad.

```
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: tiempo
Enviando: Fri Jun 02 17:20:59 GMT-05:00 2000 a iec23.utm.mx al puerto 1188.

Recibiendo un datagrama desde iec23.utm.mx puerto 1188.
Contenido del datagrama: salida
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*:

```
$ java GetTimeApp

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:58 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
El datagrama contiene los sig. datos: Fri Jun 02 17:20:59 GMT-05:00 2000

Envia peticion de tiempo a iec23 al puerto 2345.
Recibiendo un datagrama desde iec23.utm.mx at port 2345.
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.

51.2. 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("N£mero 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        outStream.writeBytes(sendLine);
90        outStream.write(13);
91        outStream.write(10);
92        outStream.flush();
93        break;
94       case 'R':
95        int inByte;
96        System.out.print("***");
97        while ((inByte = inStream.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 235. Ejemplo de uso de la clase Socket. *PortTalkApp*.

51.3. 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 conexion 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      outStream.write((byte)outLine.charAt(i));
29     outStream.write(13);
30     outStream.write(10);
31     outStream.flush();
32     System.out.println("Enviado: "+outLine);
33    } while(!finished);
34    inStream.close();
```

```
35     outStream.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 236. Ejemplo de uso de la clase *ServerSocket*.

51.4. 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 237. Ejemplo de escucha de un socket UDP. *TimeServerApp*

51.5. 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 bufferLength = 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("excepci¢n: IOException");
43   }
44  }
45 }
```

Listing 238. 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 52

Programación en Red con Java (2)

Una vez analizados los conceptos básicos de programación en red en el material pasa do, revisaremos unas clases complementarias y veremos algunos ejemplos de aplicaciones cliente/servidor.

52.1. 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].

[1] Este 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.

52.2. 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 envié 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.

52.2.1. Programas cliente

Se muestran a continuación ejemplos de programas cliente básicos.

Ejemplo:

```
1  //Programa que almacena páginas html
2  import java.util.Vector;
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  }
```

```
}

class CapturaHtml {
 String urlList = "urlList.txt";
 Vector URLs = new Vector();
 Vector fileNames = new Vector();
 public CapturaHtml() {
  super();
 }
 public void getURLList() {
  try {
   BufferedReader inStream = new BufferedReader(new FileReader(urlList));
   String inLine;
   while((inLine = inStream.readLine()) != null) {
    inLine = inLine.trim();
    if(!inLine.equals("")) {
     int tabPos = inLine.lastIndexOf('\t');
     String url = inLine.substring(0,tabPos).trim();
     String fileName = inLine.substring(tabPos+1).trim();
     URLs.addElement(url);
     fileNames.addElement(fileName);
    }
   }
  }catch(IOException ex){
   error("Error leyendo "+urlList);
  }
 }
 public void run() {
  getURLList();
  int numURLs = URLs.size();
  for(int i=0;i<numURLs;++i)
   captuURL((String) URLs.elementAt(i),(String) fileNames.elementAt(i));
  System.out.println("Ok.");
 }
 public void captuURL(String urlName,String fileName) {
  try{
   URL url = new URL(urlName);
   System.out.println("Obteniendo "+urlName+"...");
   File outFile = new File(fileName);
   PrintWriter outStream = new PrintWriter(new FileWriter(outFile));
   BufferedReader inStream = new BufferedReader(
    new InputStreamReader(url.openStream()));
```

```
53     String line;
54     while ((line = inStream.readLine())!= null) outStream.println(line);
55     inStream.close();
56     outStream.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 239. 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 cliente[2].

Ejemplo:

```
1  //Ejemplo de  cliente de SMTP
2  import java.io.*;
3  import java.net.*;
```

[2]El 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.

```
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 mensage
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 240. 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
2  import java.applet.Applet;
3  import java.awt.*;
4  import java.net.*;
5  import java.io.*;
6
7  public class TicTacToeClient extends Applet
```

```
                              implements Runnable {
   TextField id;
   TextArea display;
   Panel boardPanel, panel2;
   Square board[][], currentSquare;
   Socket connection;
   DataInputStream input;
   DataOutputStream output;
   Thread outputThread;
   char myMark;


   public void init()
   {
      setLayout( new BorderLayout() );
      display = new TextArea( 4, 30 );
      display.setEditable( false );
      add( "South", display );

      boardPanel = new Panel();
      boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );
      board = new Square[ 3 ][ 3 ];

      for ( int row = 0; row < board.length; row++ )
         for (int col = 0; col < board[row].length; col++ ) {
            board[ row ][ col ] = new Square();
            boardPanel.add( board[ row ][ col ] );
         }
      id = new TextField();
      id.setEditable( false );
      add( "North", id );

      panel2 = new Panel();
      panel2.add( boardPanel );
      add( "Center", panel2 );
   }

   public void start()
   {
      try {
         connection =
            new Socket( InetAddress.getLocalHost(), 5000 );
```

```
50          input = new DataInputStream(
51             connection.getInputStream() );
52          output = new DataOutputStream(
53             connection.getOutputStream() );
54       }
55       catch ( IOException e ) {
56          e.printStackTrace();
57       }
58
59       outputThread = new Thread( this );
60       outputThread.start();
61    }
62
63    public boolean mouseUp( Event e, int x, int y )
64    {
65       for ( int row = 0; row < board.length; row++ ) {
66          for (int col = 0; col < board[row].length; col++ ) {
67             try {
68                if ( e.target == board[ row ][ col ] ) {
69                   currentSquare = board[ row ][ col ];
70                   output.writeInt( row * 3 + col );
71                }
72             }
73             catch ( IOException ie ) {
74                   ie.printStackTrace();
75             }
76          }
77       }
78       return true;
79    }
80
81    public void run()
82    {
83       try {
84          myMark = input.readChar();
85          id.setText( "Eres el jugador \"" + myMark + "\"" );
86       }
87       catch ( IOException e ) {
88          e.printStackTrace();
89       }
90
91       // Recibir mensajes
```

```
92          while ( true ) {
93             try {
94                String s = input.readUTF();
95                processMessage( s );
96             }
97             catch ( IOException e ) {
98                e.printStackTrace();
99             }
100         }
101      }
102
103
104      public void processMessage( String s )
105      {
106         if ( s.equals( "Movida valida." ) ) {
107            display.appendText( "Movida valida, espere un momento.\n" );
108            currentSquare.setMark( myMark );
109            currentSquare.repaint();
110         }
111         else if ( s.equals( "Movida invalida, intente de nuevo" ) ) {
112            display.appendText( s + "\n" );
113         }
114         else if ( s.equals( "El oponente movio" ) ) {
115            try {
116               int loc = input.readInt();
117
118               done:
119               for ( int row = 0; row < board.length; row++ )
120                  for ( int col = 0;
121                        col < board[ row ].length; col++ )
122                     if ( row * 3 + col == loc ) {
123                        board[ row ][ col ].setMark(
124                           ( myMark == 'X' ? 'O' : 'X' ) );
125                        board[ row ][ col ].repaint();
126                        break done;
127                     }
128               display.appendText(
129                  "El oponente movio. Es tu turno.\n" );
130            }
131            catch ( IOException e ) {
132               e.printStackTrace();
133            }
```

```
134         }
135         else {
136            display.appendText( s + "\n" );
137         }
138      }
139   }
140
141   class Square extends Canvas {
142      char mark;
143
144      public Square()
145      {
146         resize ( 30, 30 );
147      }
148
149      public void setMark( char c ) { mark = c; }
150
151      public void paint( Graphics g )
152      {
153         g.drawRect( 0, 0, 29, 29 );
154         g.drawString( String.valueOf( mark ), 11, 20 );
155      }
156   }
```

Listing 241. Ejemplo - Cliente de TicTacToe.

52.2.2. 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  }
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 242. Ejemplo de un servidor genérico multihilado.

El programa servidor del gato, el cual lógicamente debe ejecutarse antes que los clien-

tes 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 gano 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         for ( int i = 0; i < players.length; i++ )
107            players[ i ].stop();
108
109         System.exit( 0 );
110      }
111
112      return super.handleEvent( event );
113   }
114
115   public static void main( String args[] )
116   {
117      TicTacToeServer game = new TicTacToeServer();
118
119      game.execute();
120   }
121}
```

```
122
123 class Player extends Thread {
124    Socket connection;
125    DataInputStream input;
126    DataOutputStream output;
127    TicTacToeServer control;
128    int number;
129    char mark;
130
131    public Player( Socket s, TicTacToeServer t, int num )
132    {
133       mark = ( num == 0 ? 'X' : 'O' );
134
135       connection = s;
136
137       try {
138          input = new DataInputStream(
139                      connection.getInputStream() );
140          output = new DataOutputStream(
141                      connection.getOutputStream() );
142       }
143       catch( IOException e ) {
144          e.printStackTrace();
145          System.exit( 1 );
146       }
147
148       control = t;
149       number = num;
150    }
151
152    public void otherPlayerMoved( int loc )
153    {
154       try {
155          output.writeUTF( "El oponente movio" );
156          output.writeInt( loc );
157       }
158       catch ( IOException e ) {}
159    }
160
161    public void run()
162    {
163       boolean done = false;
```

```
164
165         try {
166            control.display( "Jugador " +
167               ( number == 0 ? 'X' : 'O' ) + " conectado" );
168            output.writeChar( mark );
169            output.writeUTF( "Jugador " +
170               ( number == 0 ? "X conectado\n" :
171                               "O conectado, espere un momento\n" ) );
172
173            if ( control.getNumberOfPlayers() < 2 ) {
174               output.writeUTF( "Esperando otro jugador" );
175
176               while (control.getNumberOfPlayers() < 2 )
177                  ;
178
179               output.writeUTF(
180                  "Ya se conectó otro jugador. Es tu turno." );
181            }
182
183            while ( !done ) {
184               int location = input.readInt();
185
186               if ( control.validMove( location, number ) ) {
187                  control.display( "pos: " + location );
188                  output.writeUTF( "Movida válida." );
189               }
190               else
191                  output.writeUTF( "Movida inválida, intente de nuevo" );
192
193               if ( control.gameOver() )
194                  done = true;
195            }
196            connection.close();
197         }
198         catch( IOException e ) {
199            e.printStackTrace();
200            System.exit( 1 );
201         }
202      }
203   }
```

Listing 243. Ejemplo - Servidor de TicTacToe.

Capítulo 53

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.

53.1. 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].

[1] Este 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.

53.2. 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 envié 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.

53.2.1. Programas cliente

Se muestran a continuación ejemplos de programas cliente básicos.

Ejemplo:

```
1  //Programa que almacena páginas html
2  import java.util.Vector;
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   Vector URLs = new Vector();
16   Vector fileNames = new Vector();
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.addElement(url);
31       fileNames.addElement(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((String) URLs.elementAt(i),(String) fileNames.elementAt(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 outStream = new PrintWriter(new FileWriter(outFile));
51     BufferedReader inStream = new BufferedReader(
52      new InputStreamReader(url.openStream()));
```

```
53    String line;
54    while ((line = inStream.readLine())!= null) outStream.println(line);
55    inStream.close();
56    outStream.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 244. 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 cliente[2].

Ejemplo:

```
1 //Ejemplo de  cliente de SMTP
2 import java.io.*;
3 import java.net.*;
```

[2]El 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.

```
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 mensage
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 245. 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
2  import java.applet.Applet;
3  import java.awt.*;
4  import java.net.*;
5  import java.io.*;
6
7  public class TicTacToeClient extends Applet
```

```
                              implements Runnable {
   TextField id;
   TextArea display;
   Panel boardPanel, panel2;
   Square board[][], currentSquare;
   Socket connection;
   DataInputStream input;
   DataOutputStream output;
   Thread outputThread;
   char myMark;


   public void init()
   {
      setLayout( new BorderLayout() );
      display = new TextArea( 4, 30 );
      display.setEditable( false );
      add( "South", display );

      boardPanel = new Panel();
      boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );
      board = new Square[ 3 ][ 3 ];

      for ( int row = 0; row < board.length; row++ )
         for (int col = 0; col < board[row].length; col++ ) {
            board[ row ][ col ] = new Square();
            boardPanel.add( board[ row ][ col ] );
         }
      id = new TextField();
      id.setEditable( false );
      add( "North", id );

      panel2 = new Panel();
      panel2.add( boardPanel );
      add( "Center", panel2 );
   }

   public void start()
   {
      try {
         connection =
            new Socket( InetAddress.getLocalHost(), 5000 );
```

```
50          input = new DataInputStream(
51             connection.getInputStream() );
52          output = new DataOutputStream(
53             connection.getOutputStream() );
54       }
55       catch ( IOException e ) {
56          e.printStackTrace();
57       }
58
59       outputThread = new Thread( this );
60       outputThread.start();
61    }
62
63    public boolean mouseUp( Event e, int x, int y )
64    {
65       for ( int row = 0; row < board.length; row++ ) {
66          for (int col = 0; col < board[row].length; col++ ) {
67             try {
68                if ( e.target == board[ row ][ col ] ) {
69                   currentSquare = board[ row ][ col ];
70                   output.writeInt( row * 3 + col );
71                }
72             }
73             catch ( IOException ie ) {
74                   ie.printStackTrace();
75             }
76          }
77       }
78       return true;
79    }
80
81    public void run()
82    {
83       try {
84          myMark = input.readChar();
85          id.setText( "Eres el jugador \"" + myMark + "\"" );
86       }
87       catch ( IOException e ) {
88          e.printStackTrace();
89       }
90
91       // Recibir mensajes
```

```
92          while ( true ) {
93             try {
94                String s = input.readUTF();
95                processMessage( s );
96             }
97             catch ( IOException e ) {
98                e.printStackTrace();
99             }
100         }
101      }
102
103
104      public void processMessage( String s )
105      {
106         if ( s.equals( "Movida valida." ) ) {
107            display.appendText( "Movida valida, espere un momento.\n" );
108            currentSquare.setMark( myMark );
109            currentSquare.repaint();
110         }
111         else if ( s.equals( "Movida invalida, intente de nuevo" ) ) {
112            display.appendText( s + "\n" );
113         }
114         else if ( s.equals( "El oponente movio" ) ) {
115            try {
116               int loc = input.readInt();
117
118               done:
119               for ( int row = 0; row < board.length; row++ )
120                  for ( int col = 0;
121                        col < board[ row ].length; col++ )
122                     if ( row * 3 + col == loc ) {
123                        board[ row ][ col ].setMark(
124                           ( myMark == 'X' ? 'O' : 'X' ) );
125                        board[ row ][ col ].repaint();
126                        break done;
127                     }
128               display.appendText(
129                  "El oponente movio. Es tu turno.\n" );
130            }
131            catch ( IOException e ) {
132               e.printStackTrace();
133            }
```

```
      }
      else {
         display.appendText( s + "\n" );
      }
   }
}

class Square extends Canvas {
   char mark;

   public Square()
   {
      resize ( 30, 30 );
   }

   public void setMark( char c ) { mark = c; }

   public void paint( Graphics g )
   {
      g.drawRect( 0, 0, 29, 29 );
      g.drawString( String.valueOf( mark ), 11, 20 );
   }
}
```

Listing 246. Ejemplo - Cliente de TicTacToe.

53.2.2. 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:

```
//Ejemplo de un servidor genérico multihilado
import java.net.*;
import java.io.*;
import java.util.*;

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  }
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 247. Ejemplo de un servidor genérico multihilado.

El programa servidor del gato, el cual lógicamente debe ejecutarse antes que los clien-

tes 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 gano 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         for ( int i = 0; i < players.length; i++ )
107            players[ i ].stop();
108
109         System.exit( 0 );
110      }
111
112      return super.handleEvent( event );
113   }
114
115   public static void main( String args[] )
116   {
117      TicTacToeServer game = new TicTacToeServer();
118
119      game.execute();
120   }
121}
```

```
122
123 class Player extends Thread {
124    Socket connection;
125    DataInputStream input;
126    DataOutputStream output;
127    TicTacToeServer control;
128    int number;
129    char mark;
130
131    public Player( Socket s, TicTacToeServer t, int num )
132    {
133       mark = ( num == 0 ? 'X' : 'O' );
134
135       connection = s;
136
137       try {
138          input = new DataInputStream(
139                      connection.getInputStream() );
140          output = new DataOutputStream(
141                      connection.getOutputStream() );
142       }
143       catch( IOException e ) {
144          e.printStackTrace();
145          System.exit( 1 );
146       }
147
148       control = t;
149       number = num;
150    }
151
152    public void otherPlayerMoved( int loc )
153    {
154       try {
155          output.writeUTF( "El oponente movio" );
156          output.writeInt( loc );
157       }
158       catch ( IOException e ) {}
159    }
160
161    public void run()
162    {
163       boolean done = false;
```

```
164
165         try {
166            control.display( "Jugador " +
167               ( number == 0 ? 'X' : 'O' ) + " conectado" );
168            output.writeChar( mark );
169            output.writeUTF( "Jugador " +
170               ( number == 0 ? "X conectado\n" :
171                               "O conectado, espere un momento\n" ) );
172
173            if ( control.getNumberOfPlayers() < 2 ) {
174               output.writeUTF( "Esperando otro jugador" );
175
176               while (control.getNumberOfPlayers() < 2 )
177                  ;
178
179               output.writeUTF(
180                  "Ya se conectó otro jugador. Es tu turno." );
181            }
182
183            while ( !done ) {
184               int location = input.readInt();
185
186               if ( control.validMove( location, number ) ) {
187                  control.display( "pos: " + location );
188                  output.writeUTF( "Movida válida." );
189               }
190               else
191                  output.writeUTF( "Movida inválida, intente de nuevo" );
192
193               if ( control.gameOver() )
194                  done = true;
195            }
196            connection.close();
197         }
198         catch( IOException e ) {
199            e.printStackTrace();
200            System.exit( 1 );
201         }
202      }
203   }
```

Listing 248. Ejemplo - Servidor de TicTacToe.

Referencias

Apéndice A. Herramientas adicionales sugeridas

Capítulo 54

Temas pendientes

54.0.1. Decoradores en Python

¿Qué son?

Los decoradores (*decorators*) son una forma de envolver una función, como envolver un paquete con papel de regalo. Estos añaden funcionalidad a la función siendo decorada. Toman otra función como entrada y devuelven una nueva función modificada.

¿Por qué usarlos?

Cuando se necesita código en nuestros programas que se va a ejecutar frecuentemente, podemos poner ese código en un decorador.

Pueden ser utilizados para aplicar cambios en una función existente sin tener que modificar su código. Esto separa el código adicional de la función principal y, si el código adicional no se necesita más, podemos simplemente remover el decorador.

¿Cómo se escriben los decoradores?

Se escribe una función decorada que toma a otra función cómo argumento. Dentro de la función decoradora se escribe una función interna que contiene el código adicional, ejecuta el argumento de la función. son aplicados a una función utilizando el símbolo "@" seguido del nombre del decorador. Los decoradores son una forma de "meta-programaciónza que permiten modificar el comportamiento de una función en tiempo de ejecución. Se regresa la función interna.

Listing 249. Ejemplo de uso de decoradores.

En este ejemplo se define la función decoradora *registrar_hora* que toma como parámetro una función y devuelve una nueva función *envoltorio* que imprime la hora actual y el nombre de la función que se esta ejecutando antes de llamar a la función original y devolver su resultado. Se aplica el decorador a la función *mi_funcion* usando el símbolo "@" antes

de su declaración y se ejecuta la función *mi_funcion*, la cual imprimirá ”Hola mundo!” y también la hora en la que se ejecutó la función.

Otros ejemplos:

Listing 250. Ejemplo de decorador para medir tiempo de ejecución de una función.

En el ejemplo entarior, la función decoradora *temporizador* toma otra función como argumento y devuelve una nueva función *envoltorio* que mide el tiempo de ejecución de la función original y lo imprime antes de devolver el resultado. La función *funcion_lenta* simula una tarea que dura 1 segundo. Al llamar a esta función se imprimirá ”La función funcion_lenta tardó 1.0 segundos en ejecutarse.”

Otro ejemplo seria el siguiente:

Listing 251. Ejemplo de decorador que verifica un argumento.

En este ejemplo se define la función decoradora *asegurar_primer_arg_es* que toma un valor y devuelve una nueva función *interno* que a su vez toma otra función como argumento y devuelve una nueva función *envoltorio* que verifica si el primer argumento de la función original es igual al valor especificado, si no es así devuelve un mensaje de error, si es así ejecuta la función original. Se aplica este decorador a la función *comida_fav* y se llama a esta función con diferentes argumentos, si el primer argumento es ”pizza” se ejecuta normalmente, si no se imprime el mensaje de error.

Decoradores predefinidos más comunes[1]

Algunos decoradores usados con frecuencia:

- dataclass. Éste decorador se utiliza para crear clases de datos de manera rápida y sencilla. Al aplicar @dataclass a una clase, Python automáticamente genera ciertos métodos para la clase.
- property. Éste decorador se utiliza para definir un método como una propiedad de una clase, lo que permite acceder al método como si fuera un atributo.
- staticmethod. Éste decorador se utiliza para definir un método estático, que no tiene acceso a los atributos de la clase o de la instancia.
- classmethod. Éste decorador se utiliza para definir un método como un método de clase, lo que permite acceder al método a través de la clase en lugar de a través de una instancia.

[1] Partes de esta sección fueron desarrollados con apoyo de ChatGPT

- abstractmethod. Éste decorador se utiliza para definir un método abstracto en una clase abstracta. Los métodos abstractos son métodos que deben ser implementados por las subclases.

Al final, un decorador es únicamente una función con código en ésta.

@dataclass

@dataclass es un decorador de Python 3.7 que es parte de la biblioteca estándar, se utiliza para crear clases de datos de manera rápida y sencilla.

Al aplicar @dataclass a una clase, Python automáticamente genera ciertos métodos para la clase, como las funciones especiales *init*, *repr* y *eq* , así como también agrega la clase al módulo *dataclasses*.

Además, permite especificar los atributos de clase mediante la creación de variables de instancia con el decorador *field*. Esto proporciona una manera fácil de especificar los atributos de clase y también proporciona una forma de especificar opciones adicionales como el valor predeterminado, si el campo es inmutable, etc.

Un ejemplo de uso de @dataclass se muestra a continuación:

Listing 252. Ejemplo de @dataclass.

En este ejemplo, se utiliza @dataclass para definir la clase Persona, y se utilizan los tipos de datos básicos de Python para especificar los atributos de la clase. Al imprimir las instancias de Persona, se puede ver que @dataclass ha generado automáticamente un método de representación legible de la instancia.

Otro ejemplo se muestre enseguida:

Listing 253. Ejemplo adicional de @dataclass .

En este ejemplo se tiene una clase Alumno la cual tiene lo campos nombre, edad y notas. El campo notas se inicializa con una lista vacía.

Al crear una instancia de la clase Alumno (alumno1 = Alumno(”Juan”)), se genera automáticamente un método especial *init* que asigna los valores de los atributos nombre, edad y notas a los valores especificados al crear la instancia. También se genera un método especial repr que permite generar una representación legible de la instancia y un método especial eq que permite comparar dos instancias de la clase para ver si son iguales. En este caso también se agregó un método mostrar_notas para mostrar las notas del alumno

Se uso la función field(). field() es una función decoradora incluida en el módulo dataclasses de Python. Se utiliza para especificar ciertas opciones para los campos de una clase de datos. Cuando se aplica a un atributo de una clase, indica que ese atributo es un campo y se debe tratar de manera especial por el decorador dataclass.

Los argumentos opcionales que se pueden pasar a la función field() son:

default: especifica un valor por defecto para el campo. default_factory : especifica una función que se ejecuta
indica si el campo debe ser incluido en el constructor de la clase. Por defecto es True. repr : indica si el campo debe ser
indica si el campo debe ser comparado al comparar dos instancias de la clase. Por defecto es True. hash :
indica si el campo debe ser incluido en el cálculo del valor hash de la instancia. Por defecto es True.

field() es una forma de personalizar cómo se comporta la clase con el decorador dataclass. Sin embargo, no es necesario utilizar field() en cada atributo de una clase decorada con dataclass, ya que por defecto se consideran todos los atributos como campos con las opciones por defecto.

Por otro lado, al momento de usar el decorador, es posible pasar algunos parámetros. Los parámetros posibles que se pueden pasar a la decoración @dataclass son:

init: indica si se debe generar automáticamente un constructor para la clase. Por defecto es True. eq: indica si se debe generar automáticamente el método $eq_{para la clase. Por defecto es True. order: indica si se debe generar automática}$

Un ejemplo usando algunos de los parámetros de dataclass:

Listing 254. Ejemplo .

En este ejemplo se está utilizando el parámetro *frozen=True* para congelar la clase, lo que significa que no se pueden modificar los valores de los atributos una vez que se ha creado una instancia. También se está utilizando el parámetro *order=True* para habilitar la comparación entre instancias de la clase. Además se utiliza *asdict* para generar un diccionario a partir de una instancia de la clase.

54.0.2. Python Generators

¿Qué son los generadores?

Los generadores nos permiten escribir funciones eficientes que puedan actuar como iteradores, por ejemplo, pueden usarse en un ciclo *for*. Cualquier función que regresa un iterador puede convertirse a un generador usando la palabra reservada *yield*.

¿Cómo escribir un generador?

Cualquier función que regresa un iterador puede convertirse a un generador usando la palabra reservada *yield*.

En el primer ejemplo, regresamos una lista conteniendo cubos de números hasta el 50 al mismo tiempo.

¡ejemplo¿

En el segundo ejemplo cedemos (*yield*) el siguiente cubo bajo demanda

¡ejemplo¿

¿Por qué usar generadores? Son limpios y hacen uso eficiente de memoria.

Del ejemplo anterior imaginemos la diferencia entre tener que crear y mantener una lista de cubos de 50 números al mismo tiempo contra únicamente regresar el siguiente valor bajo demanda.

¿Cuando usar generadores?

Cuando sea necesario ahorrar memoria.
Si se está inseguro si todos los elementos en un iterador no van a ser usados.

54.0.3. Python Sys y OS

54.0.4. Python- Shelves

54.0.5. Python - Persistence

54.0.6. Python - GUI - PySimpleGUI

https://twitter.com/driscollis/status/1570115002249498646

54.0.7. Python - GUI - tkinter

54.1. Envío múltiple (Multiple dispatch

6 Encapsulamiento

54.2. Principios de encapsulamiento

54.3. Patrón de diseño proxy

54.4. Modelado y construcción de componentes