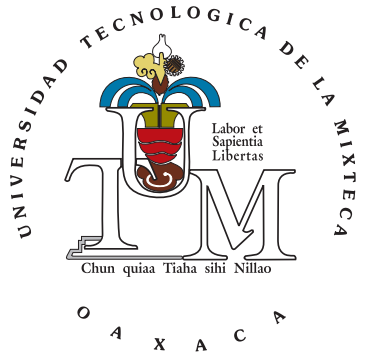


Notas de Programación Orientada a Objetos con Java

POO Handbook 2025

Carlos Alberto Fernández y Fernández

Instituto de Computación
Universidad Tecnológica de la Mixteca

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

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

– Dante Alighieri, La Divina Comedia

27 de noviembre de 2025

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

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

Cuadro 1.1. Comparación de lenguajes de programación

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

1.1.1 IDEs

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

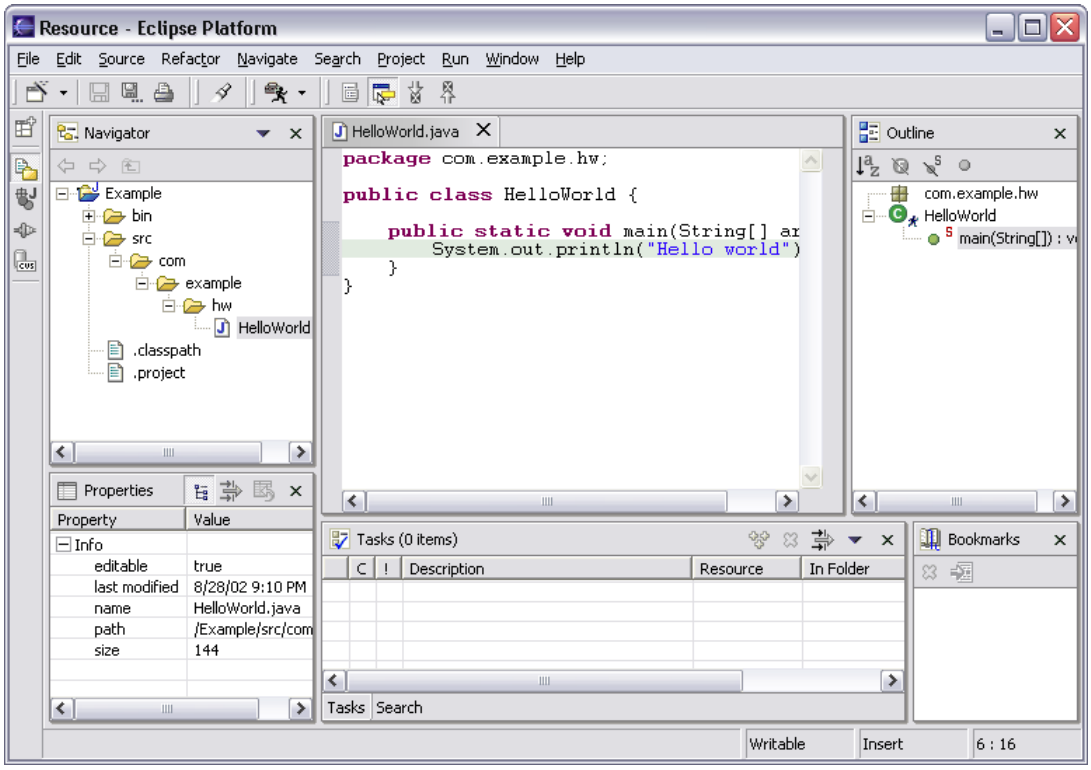
Eclipse

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

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

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

Trabaja bajo "*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

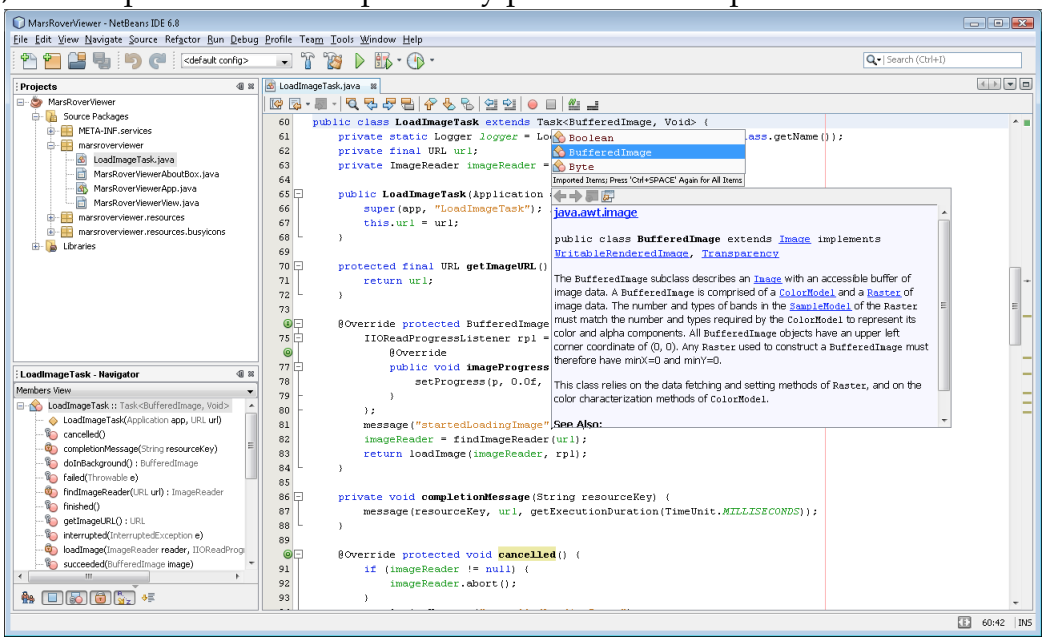
Mono

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.

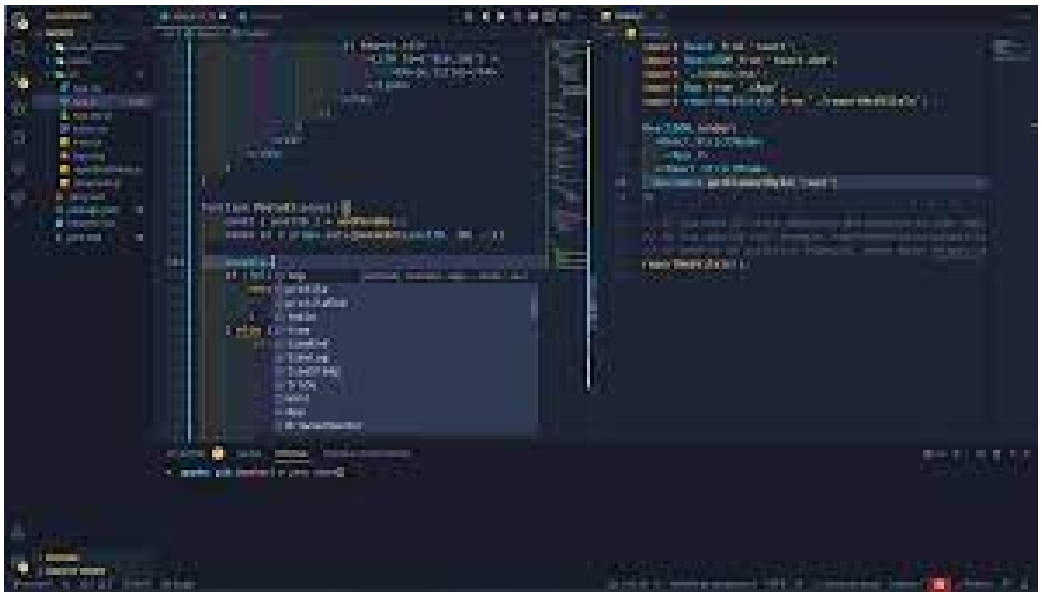


Soporta una variedad de lenguajes y tecnologías como C/C++, Java (SE, EE, ME), Ruby y PHP. NetBeans destaca por integrar de manera nativa un diseñador visual de interfaces gráficas (GUI Builder), útil en el desarrollo rápido de aplicaciones de escritorio.

Visual Studio Code

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

[1] https://code.visualstudio.com/



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

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

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

Algunos editores ligeros

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

- Geany. [2]
- Sublime Text. [3]
- Atom. [4]

[2] https://www.geany.org/
[3] https://www.sublimetext.com/
[4] https://atom.io/

- 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.

[5] http://brackets.io/

Comparación de IDEs

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

Cuadro 1.2. Breve comparación de IDEs

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

1.2 Lenguajes estáticos y dinámicos

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

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

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

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

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

$g++ -o foo.o foo.cpp$

o, desde el compilador de C:

$gcc -o foo.o foo.cpp -lstdc++$

Capítulo 2

Introducción a Java

2.1 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.

2.2 Características de diseño

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

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

Simple y familiar

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

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

Orientado a objetos

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

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

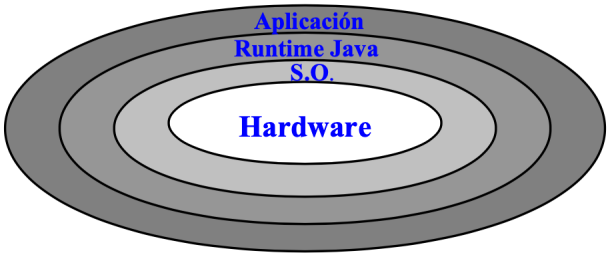


Fig. 2.1. Máquina virtual de Java y su ejecución sobre una plataforma

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

Cumple con los 4 requerimientos de Wegner [**?**]:

OO = abstracción + clasificación + polimorfismo + herencia

Independiente de la plataforma

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

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

Si el sistema de *runtime* o máquina virtual está disponible para una plataforma específica, entonces una aplicación puede ejecutarse sin necesidad de un trabajo de programación adicional. Ver figura 2.1.

Portable

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

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

[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.

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.

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

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

2.3 Diferencias entre Java y C++

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

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

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

Además:

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

2.4 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*.

[3] *Garbage Collector*

2.5 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].

2.6 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.

[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.

[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.

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.

2.7 "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 1. 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;
5
6 int main(){
7     cout << "Hola Mundo!" << endl;
8     return 0;
9 }
```

Listing 2. Hola, Mundo en C++.

[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.

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
```

2.8 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.

2.8.1 Comentarios

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

Existen tres tipos de comentarios en Java:

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

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

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

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

910

```
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/

2.8.2 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 2.1. 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).

2.8.3 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.

2.8.4 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.

2.8.5 Operadores

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

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

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

Cuadro 2.2. Operadores aritméticos Java

Operadores aritméticos:

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

Como en C/C++, los operadores unarios ++ y – realizan un incremento y un decremento respectivamente. Estos operadores admiten notación prefija y postfija. Ver Cuadro 2.2

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

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

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

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

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 2.3. Operadores relacionales en Java

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

Ejemplo:

```
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 2.3

Ejemplo:

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

Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve *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 2.4. Operadores lógicos en Java

```
15          }
16  }
```

Listing 3. Ejemplo de operadores relacionales en Java.

Operadores lógicos:

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso. Ver Cuadro 2.4

Ejemplo:

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

Listing 4. Ejemplo de operadores lógicos en Java.

Operadores de asignación:

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

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

Cuadro 2.5. Operadores de asignación en Java

```
op1 = Expresión;
```

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

Además del operador de asignación existen unas abreviaturas, como en C/C++, cuando el operando que aparece a la izquierda del símbolo de asignación también aparece a la derecha del mismo. Ver Cuadro 2.5

Precedencia de operadores en Java

La precedencia (ver Cuadro 2.6)indica el orden en que es resuelta una expresión, la siguiente lista muestra primero los operadores de mayor precedencia.

2.8.6 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	$\hat{}\hat{}\ >$
Relacionales	$<><=>=$
Igualdad y desigualdad	$==!=$
AND a nivel de bits	&
OR a nivel de bits	$\vert$
AND lógico	&&
OR lógico	$\vert\vert$
Condicional terciaria	? :
Asignación	$=+=-=*=/=\%=\&=\vert=\check{}=\acute{}=\check{}>=$

Cuadro 2.6. 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 2.7

Literales de tipo *String*

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

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

Ejemplo:

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

Cuadro 2.7. 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.''
```

2.8.7 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 5. 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 6. Ejemplo de uso de la **expresión** switch en Java.

Ciclos

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

Existen tres construcciones para estas estructuras de repetición:

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

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

Ciclo for

Sintaxis

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

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

- La cláusula condición es una expresión lógica, que se evalúa al inicio de cada nueva iteración del ciclo. En el momento en que dicha expresión se evalúe a falso, se dejará de ejecutar el ciclo y el control del programa pasará a la siguiente instrucción (a continuación del ciclo *for*).

- La cláusula incremento es una instrucción que se ejecuta en cada iteración del ciclo como si fuera la última instrucción dentro del bloque de instrucciones. Generalmente se trata de una instrucción de incremento o decremento de alguna variable.

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

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

Ejemplo:

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

Listing 7. Ejemplo Fibonacci con ciclo *for*.

Ciclo do-while

Sintaxis

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

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

Ejemplo:

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

Listing 8. Ejemplo Fibonacci con ciclo *do-while*.

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

se ejecuta una vez por lo menos.

Ciclo while

Sintaxis

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

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

Ejemplo:

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

Listing 9. 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 10. 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 11. 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.

2.8.8 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:

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

Listing 12. 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 13. Ejemplo de uso de arreglo 2.

2.8.9 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 14. Ejemplo de enumeración en Java.

2.8.10 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 15. 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]:

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

Listing 16. 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.

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

Argumentos de cantidad variable

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

Ejemplo:

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

Listing 17. 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
```

[14]Código original de [**?**]

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

Listing 18. Ejemplo de entrada de argumentos de cantidad variable.

2.8.11 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 3

Introducción a la programación orientada a objetos [? ?]

3.1 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.

3.2 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.

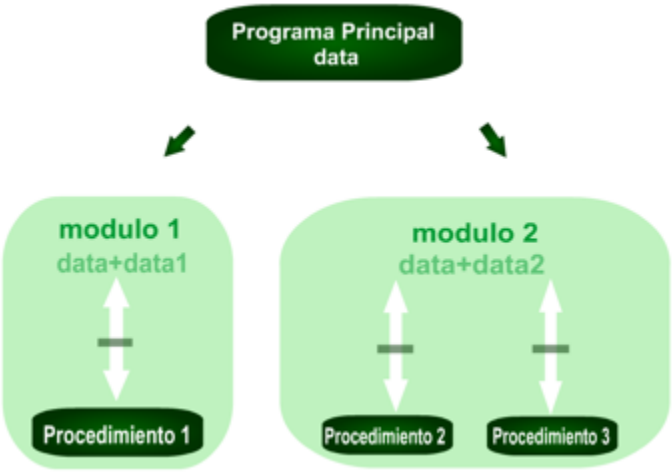
3.3 Programación modular

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.



3.4 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.

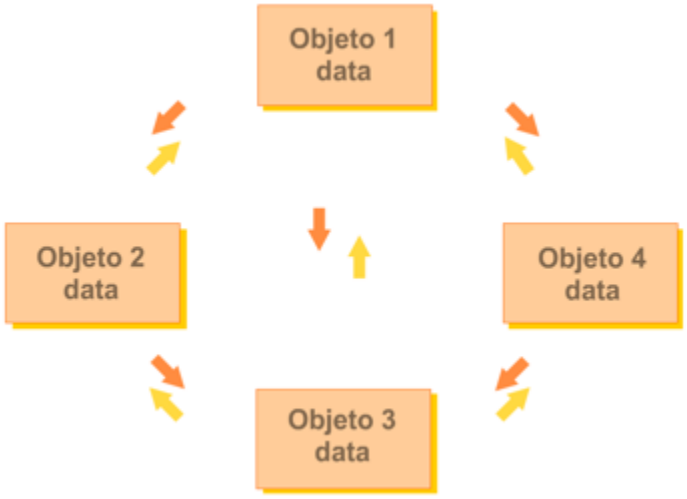
En la programación orientada a objetos, la estructura se organiza por los datos. Se escogen las representaciones de datos que mejor se ajusten a tus requerimientos. Por consecuencia, los programas se estructuran por los datos más que por las operaciones.

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

3.5 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.

3.6 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.

3.6.1 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.

3.6.2 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.

3.7 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.

3.8 Lenguajes de programación orientada a objetos

Simula I (1967) fue originalmente diseñado para problemas de simulación y fue el primer lenguaje en el cual los datos y procedimientos estaban unificados en una sola entidad. Su sucesor **Simula** (1973) , derivó definiciones formales a los conceptos de objetos y clase.

Simula sirvió de base a una generación de lenguajes de programación orientados a objetos. Es el caso de **C++** (1985), **Eiffel** (1986) y **Beta**. (1987)

Ada (1983), se derivan de conceptos similares, e incorporan el concepto de jerarquía de herencia. **CLU** -*clusters*- (1986) también incorpora herencia.

Cuadro 3.1. Características de LPOO

	Ada 95	Eiffel	Smalltalk	C++	Java
Paquetes	Sí	No	No	No	Sí
Herencia	Simple	Múltiple	Simple	Múltiple	Simple
Control de tipos	Fuerte	Fuerte	Sin tipos	Fuerte	Fuerte
Enlace	Dinámico	Dinámico	Dinámico	Dinámico	Dinámico
Concurrencia	Sí	No	No	No	Sí
Recolección de basura	No	Sí	Sí	No	Sí
Afirmaciones	No	Sí	No	No	Sí*
Persistencia	No	No	No	No	No
Generecidad	Sí	Sí	Sí	Sí	Sí*

*Afirmaciones y generecidad en Java fueron incluidos a partir de la versión 5 (1.5) del lenguaje.

Smalltalk es descendiente directo de **Simula**, generaliza el concepto de objeto como única entidad manipulada en los programas. Existen tres versiones principales: **Smalltalk-72**, introdujo el paso de mensajes para permitir la comunicación entre objetos. **Smalltalk-76** que introdujo herencia. **Smalltalk-80** se inspira en **Lisp**.

Lisp contribuyó de forma importante a la evolución de la programación orientada a objetos.

Flavors (1986) maneja herencia múltiple apoyada con facilidades para la combinación de métodos heredados.

CLOS (1989), es el estándar del sistema de objetos de **Common Lisp**.

Los programas de programación orientada a objetos pierden eficiencia ante los lenguajes imperativos, pues al ser interpretado estos en la arquitectura *von Neumann* resulta en un **excesivo** manejo dinámico de la memoria por la constante creación de objetos, así como una fuerte carga por la división en múltiples operaciones (métodos) y su ocupación. Sin embargo se gana mucho en **comprensión** de código y **modelado** de los problemas.

3.8.1 Características de los algunos LPOO[1]

En el Cuadro 3.1 podemos ver algunas características de lenguajes de programación orientados a objetos.

[1]Lenguajes de Programación Orientada a Objetos

Capítulo 4

Abstracción de datos: Clases y objetos

4.1 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,
- ...

4.2 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 [**?**].

4.2.1 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.

4.3 Clases en Java

La definición en **Java**, de manera similar a C++, consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves.

Sintaxis

```
class <nombre_clase> {
        <cuerpo de la clase>
}
```

Ejemplo[1]:

```
1 public class Ejemplo1 {
2         int x;
3         float y;
4         void fun(int a, float b) {
5                 x=a;
6                 y=b;
7         }
8 }
```

4.4 Miembros de una clase en Java

Los miembros en Java son esencialmente los atributos y los métodos de la clase. Ejemplo:

```
1 class Ejemplo2{
2         int i;
3         int i;    //error
4         int j;
5         int func(int, int){}
6 }
```

[1] Algunos ejemplos como este no son programas completos, sino simples ejemplos de clases. Podrán ser compilados pero no ejecutados directamente. Para que un programa corra debe contener o ser una clase derivada de *applet*, o tener un método *main*.

4.4.1 Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

La sintaxis mínima es la siguiente:

Sintaxis

```
tipo nombreAtributo;
```

Los atributos pueden ser inicializados desde su lugar de declaración:

```
tipo nombreAtributo = valor;
```

o, en el caso de variables de objetos:

```
tipo nombreAtributo = new Clase();
```

4.4.2 Métodos miembro

Un método es una operación que pertenece a una clase. No es posible declarar métodos fuera de la clase. Además, en Java no existe el concepto de método prototipo como en C++.

Sin embargo, igual que en C++, la declaración de una función ó método miembro es considerada dentro del ámbito de su clase.

La sintaxis básica para declarar a un método:

Sintaxis

```
tipoRetorno nombreMétodo ( [<parámetros>] ) {
        <instrucciones>
}
```

Un aspecto importante a considerar es que el paso de parámetros en Java es realizado exclusivamente por valor. Datos básicos y objetos son pasados por valor. Pero los objetos no son pasados realmente, se pasan las referencias a los objetos (i.e., una copia de la referencia al objeto).

4.4.3 Un vistazo al acceso a miembros

Si bien en Java existen también los miembros públicos y privados, estos tienen una sintaxis diferente a C++. En Java se define el acceso a cada miembro de manera unitaria, al contrario de la definición de acceso por grupos de miembros de C++.

Miembros públicos

Sintaxis
`public <definición de miembro>`

Miembros privados

Sintaxis
`private <definición de miembro>`

Si se omite el nivel de acceso de un miembro, se considera como *acceso de paquete*. Es decir, se tiene acceso al miembro únicamente dentro del paquete en el que la clase esta declarada[2].

Recordatorio
Es una buena costumbre de programación acceder a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo. Estos métodos de acceso y modificación comúnmente tienen el prefijo *get* y *set*, respectivamente.

Ejemplo:

```
1 class Fecha {
2         private int dia;
3         private int mes, an;
4
```

[2]Fuente: Java documentation

```
5          public boolean setDia(int d){}    //poner día
6          public int getDia()         {} //devuelve día
7          public boolean setMes(int m){}
8          public int getMes(){}
9          public boolean setAño(int a) {}
10         public int getAño() {}
11 }
```

4.5 Objetos de clase en Java

En Java todos los objetos son creados dinámicamente, por lo que se necesita reservar la memoria de estos en el momento en que se van a ocupar. El operador de Java está basado también en el de C++ y es *new*[3].

4.5.1 Asignación de memoria al objeto

El operador *new* crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. Esta referencia debe de recibirla un identificador de la misma clase de la que se haya reservado la memoria.

Sintaxis

```
<identificador> = new Clase();
```

o en el momento de declarar a la variable de objeto:

```
Clase <identificador> = new Clase();
```

El concepto de *new* va asociado de la noción de constructor, pero esta se verá más adelante, por el momento basta con adoptar esta sintaxis para poder completar ejemplos de instanciación.

Un ejemplo completo:

[3]La instrucción *new*, ya había sido usada para reservar memoria a un arreglo, ya que estos son considerados objetos.

```
1  public class Ejemplo3 {
2          public int i, j;
3
4          public static void main(String argv[]) {
5                  Ejemplo3 e3= new Ejemplo3();
6                  Ejemplo3 e1= new Ejemplo3();
7
8                  e1.i=10;
9                  e1.j=20;
10                 e3.i=100;
11                 e3.j=20;
12                 System.out.println(e1.i);
13                 System.out.println(e3.i);
14         }
15  }
```

Listing 19. Ejemplo de clase en Java.

Otro ejemplo, una estructura de cola:

```
1  class Cola{
2          private int q[];
3          private int sloc, rloc;
4
5          public void ini() {
6                  sloc=rloc=-1;
7          q=new int[10];
8          }
9
10     public boolean set(int val){
11                 if(sloc>=10){
12                         System.out.println("la cola esta llena");
13                         return false;
14                 }
15                 sloc++;
16                 q[sloc]=val;
17                 return true;
18         }
19
20         public int get(){
21         if(rloc==sloc) {
22             System.out.println("la cola esta vacia");
```

```
23              return -1;
24          }
25                  else {
26                          rloc++;
27                          return q[rloc];
28                  }
29          }
30  }
31
32  public class PruebaCola {
33          public static void main(String argv[]){
34
35                  Cola a= new Cola(); // new crea realmente el objeto
36                  Cola b= new Cola(); // reservando la memoria
37                  Cola pCola= new Cola();
38
39          //Inicializacion de los objetos
40                  a.ini();
41                  b.ini();
42                  pCola.ini();
43
44                  a.set(1);
45                  b.set(2);
46                  pCola.set(3);
47                  a.set(11);
48                  b.set(22);
49                  pCola.set(33);
50
51                  System.out.println(a.get());
52                  System.out.println(a.get());
53                  System.out.println(b.get());
54                  System.out.println(b.get());
55                  System.out.println(pCola.get());
56                  System.out.println(pCola.get());
57          }
58  }
```

Listing 20. Ejemplo de clase con una estructura de Cola simple en Java.

4.6 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 5

Polimorfismo AdHoc: Sobrecarga de operaciones

5.1 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.

5.2 Ejemplo de sobrecarga en Java

```
1 class MiClase{
2         int x;
3         public          void modifica() {
4                 x++;
5         }
6         public void modifica(int y){
7                 x=y*y;
8         }
9 }
```

Capítulo 6

Constructores y destructores

6.1 Constructores y finalizadores en Java

En Java, cuando un objeto es creado es llamada un método conocido como **constructor**, y al salir se llama a otro conocido como **finalizador**[1]. Si no se proporcionan estos métodos se asume la acción más simple.

6.1.1 Constructor

Un constructor es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato de retorno y si puede permitir la homonimia o sobrecarga, y la modificación de acceso al mismo.

Ejemplo:

```
1  public class Cola{
2          private int q[];
3          private int sloc, rloc;
4          public void put(int){ ... }
5          public int get( ){ ... }
6          //        implementación del constructor
7          public Cola ( ) {
8                  sloc=rloc=0;
9                  q= new int[100];
10                 System.out.println("Cola inicializada ");
11         }
12 }
```

El constructor se ejecuta en el momento de asignarle la memoria a un objeto, y es la razón de usar los paréntesis junto al nombre de la clase al usar la instrucción *new*:

```
Fecha f = new Fecha(10,4,2007);
```

Si no se especifica un constructor, Java incluye uno predeterminado, que asigna memoria para el objeto e inicializa las variables de instancia a valores predeterminados. Este constructor se omite si se especifica uno o más por parte del programador.

[1] En C++ no existe el concepto de finalizador, sino el de destructor, porque su tarea primordial es liberar la memoria ocupada por el objeto, cosa que no es necesario realizar en Java.

6.1.2 Finalizador

La contraparte del constructor en Java es el método *finalize* o finalizador. Este se ejecuta momentos antes de que el objeto sea destruido por el recolector de basura. El uso más común para un finalizador es liberar los recursos utilizados por el objeto, como una conexión de red o cerrar algún archivo abierto.

No es muy común utilizar un método finalizador, más que para asegurar situaciones como las mencionadas antes. El método iría en términos generales como se muestra a continuación[2]:

Sintaxis

```
protected void finalize() {
        <instrucciones>
}
```

El finalizador puede ser llamado como un método normal, inclusive puede ser sobrecargado, pero un finalizador con parámetros no puede ser ejecutado automáticamente por la máquina virtual de Java. Se recomienda evitar el definir un finalizador con parámetros.

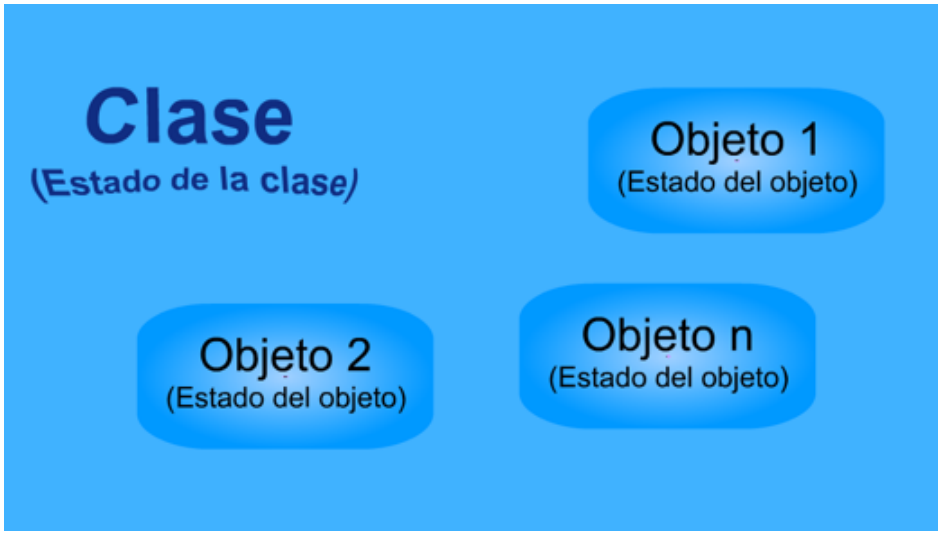
[2]No se ha mencionado el modificador *protected*. Este concepto se explicará una vez que se haya visto el manejo de herencia.

Capítulo 7

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.



7.1 Miembros estáticos en Java

Un miembro estático en Java se maneja de la misma forma que en C++. Cada uno de los objetos tiene su propio estado independiente del resto de los objetos, compartiendo al mismo tiempo un estado común al tener todos los objetos acceso al estado de la clase, el cual es único y existe de forma independiente.

Ejemplo:

```
1  public class Objeto{
2      private String nombre;
3          private static int numObjetos;
4          public Objeto(String cadena){
5                  if(cadena.length()!=0)
6                          nombre=cadena;
7                  else
8                          nombre="cadena por omision";
9                  numObjetos++;
10         }
11
12         public static int getNumObjetos(){
13                 return numObjetos;
14         }
15
16         public static void main(String argv[]) {
17                 System.out.println("Objetos: " + getNumObjetos());
18                 System.out.println("Objetos: " + Objeto.getNumObjetos());
19                 Objeto uno,dos;
20                 uno= new Objeto("");
21                 dos= new Objeto("Objeto dos");
22                 System.out.println("Objetos: " + uno.getNumObjetos());
23                 System.out.println("Objetos: " + dos.getNumObjetos());
24         }
25 }
```

Listing 21. Ejemplo de miembros estáticos en Java.

Una diferencia que se puede apreciar en este ejemplo con respecto a C++, es que no estamos obligados a inicializar el elemento estático, pues éste es inicializado automáticamente.

Capítulo 8

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.

8.1 Objetos finales en Java

Ya se mencionó en la sección de fundamentos de Java el uso de la palabra reservada *final*, la cual permite a una variable ser inicializada sólo una vez. En el caso de los objetos o referencias a los objetos el comportamiento es el mismo. Si se agrega la palabra *final* a la declaración de una referencia a un objeto, significa que la variable podrá ser inicializada una sola vez, en el momento que sea necesario.

Sintaxis

```
final <clase> <lista de identificadores de objetos>;
```

Por ejemplo:

```
final Hora h1= new Hora(9,30,20);
```

Concepto

Es importante remarcar que no es el mismo sentido de *const* en C++. Aquí lo único que se limita es la posibilidad de una variable de referencia a ser inicializada de nuevo, pero no inhibe la modificación de miembros.

Por ejemplo:

```
final Light aLight = new Light();  // variable local final
aLight.noOfWhatts = 100;         //Ok. Cambio en el edo. del objeto

aLight = new Light();         // Inválido. No se puede modificar la referencia
```

Ejemplo:

```
class Fecha {
        private int dia;
        private int mes, año;

        public Fecha(){
                dia=mes=1;
                año=1900;
        }
        public boolean setDia(int d){
                if (d >=1 && d<=31){
                        dia= d;
                        return true;
                }
                return false;

        }   //poner día
        public int getDia()         {
                return dia;
        } //devuelve día
        public boolean setMes(int m){
                if (m>=1 && m<=12){
                        mes=m;
                        return true;
                }
                return false;
        }
        public int getMes(){
                return mes;
        }

        public boolean setAño(int a) {
                if (a>=1900){
                        año=a;
                        return true;
```

```
35                  }
36                  return false;
37          }
38          public int getAño() {
39                  return año;
40          }
41  }
42
43  public class MainF {
44
45          public static void main(String[] args) {
46                  final Fecha f;
47
48                  f= new Fecha();
49
50                  f.setDia(10);
51                  f.setMes(3);
52                  f.setAño(2001);
53                  System.out.println(f.getDia()+"/"+f.getMes()+"/"+f.getAño());
54                  f= new Fecha(); //Error: la variable f es final y no puede ser reasigna
55          }
56  }
```

Listing 22. Ejemplo de objetos finales en Java.

Capítulo 9

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 10

Herencia

10.1 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

10.2 Herencia: implementación en Java

La clase de la cual se toman sus características se conoce como **superclase**; mientras que la clase que ha sido creada a partir de la clase base se conoce como **subclase**.

La herencia en Java difiere ligeramente de la sintaxis de implementación de herencia en C++.

Sintaxis

```
class <subclase> extends  <superclase> {
        //cuerpo subclase
}
```

Ejemplo:

Una clase vehículo que describe a todos aquellos objetos vehículos que viajan en carreteras. Puede describirse a partir del número de ruedas y de pasajeros. De la definición de vehículos podemos definir objetos más específicos (especializados). Por ejemplo la clase camión.

```
1  //ejemplo de herencia
2  class Vehiculo{
3          private int ruedas;
4          private int pasajeros;
5
6          public void setRuedas(int num){
7                  ruedas=num;
8          }
9
10         public int getRuedas(){
11                 return ruedas;
12         }
13
14         public void setPasajeros(int num){
15                 pasajeros=num;
16         }
17
18         public int getPasajeros(){
```

```
19                  return pasajeros;
20          }
21
22  }
23
24  //clase Camion con herencia  de Vehiculo
25  public class Camion extends Vehiculo {
26          private int carga;
27
28          public void setCarga(int num){
29                  carga=num;
30          }
31
32          public int getCarga(){
33                  return carga;
34          }
35
36          public void muestra(){
37                  // uso de métodos heredados
38                  System.out.println("Ruedas: " + getRuedas());
39                  System.out.println("Pasajeros: " + getPasajeros());
40                  // método de la clase Camion
41                  System.out.println("Capacidad de carga: " + getCarga());
42          }
43
44          public static void main(String argvs[]){
45                  Camion ford= new Camion();
46                  ford.setRuedas(6);
47                  ford.setPasajeros(3);
48                  ford.setCarga(3200);
49                  ford.muestra();
50          }
51
52  }
```

Listing 23. Ejemplo de herencia en Java.

En el programa anterior se puede apreciar claramente como una clase Vehículo hereda sus características a la subclase *Camion*, pudiendo este último aprovechar recursos que no declara en su definición.

Tema sugerido
BlueJ en apéndice A. (26.2.2)

10.2.1 Clase *Object*

En Java toda clase que se define tiene herencia implícita de una clase llamada *Object*. En caso de que la clase que crea el programador defina una herencia explícita a una clase, hereda las características de la clase *Object* de manera indirecta[1].

Ver clase *Object* en: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

10.2.2 Control de acceso a miembros en Java

Existen tres palabras reservadas para el control de acceso a los miembros de una clase: *public*, *private* y *protected*. Estas sirven para proteger los miembros de la clase en diferentes formas.

El control de acceso, como ya se vio anteriormente, se aplica a los métodos, atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

- *private*. Un miembro privado únicamente puede ser utilizado por los métodos miembro de la clase donde fue declarado. Un miembro privado no es posible que sea manejado ni siquiera en sus subclases.
- *protected*. Un miembro protegido puede ser utilizado únicamente por los métodos miembro de la clase donde fue declarado, por los métodos miembro de las clases derivadas o clases que pertenecen al mismo paquete. El acceso protegido es como un nivel intermedio entre el acceso privado y público.
- *public*. Un miembro público puede ser utilizado por cualquier método. Este es visible en cualquier lugar que la clase sea visible

Ejemplo:

```
1 //ejemplo de control de acceso
2
3 class Acceso{
4         protected int b;
5         protected int f2() {
```

[1]En este caso hay que considerar que las características de la clase *Object* pudieron haber sido modificadas a través de la jerarquía de herencia.

```
6                   return b;
7           }
8
9           private int c;
10          private int f3() {
11                  return c;
12          }
13
14          public int d, f;
15          public int f4(){
16                  return  d;
17          }
18
19  }
20
21  public class EjemploAcceso {
22
23          public static void main(String argvs[]){
24                  Acceso obj= new Acceso();
25
26                  obj.f2(); //es válido, ya que por omisión
27                  obj.b=2; //las dos clases están en el mismo paquete
28
29                  obj.c=3; //error es un atributo privado
30                  obj.f3(); //error es un método privado
31
32                  obj.d=5;
33                  obj.f4();
34          }
35  }
36
```

Listing 24. Ejemplo de control de acceso en Java.

El ejemplo anterior genera errores de compilación al tratar de acceder desde otra clase a miembros privados. Sin embargo, los miembros protegidos si pueden ser accedidos porque están considerados implícitamente dentro del mismo paquete.

10.2.3 Control de acceso de clase public en Java

Este controlador de acceso *public*, opera a nivel de la clase para que esta se vuelva visible y accesible desde cualquier lugar, lo que permitiría a cualquier otra

clase hacer uso de los miembros de la clase pública.

Sintaxis

```
public class TodosMeVen {
// definición de la clase
}
```

La omisión de este calificador limita el acceso a la clase para que solo sea utilizada por clases pertenecientes al mismo paquete.

Además, se ha mencionado que en un archivo fuente únicamente puede existir una clase pública, la cual debe coincidir con el nombre del archivo. La intención es que cada clase que tenga un objetivo importante debería ir en un archivo independiente, pudiendo contener otras clases no públicas que le ayuden a llevar a cabo su tarea.

10.2.4 Constructores de superclase

Los constructores no se heredan a las subclases. El constructor de la superclase puede ser llamado desde la clase derivada, para inicializar los atributos heredados y no tener que volver a introducir código de inicialización ya escrito en la superclase.

La llamada explícita al constructor de la superclase se realiza mediante la referencia **super** seguida de los argumentos –si los hubiera- del constructor de la clase base. La llamada a este constructor debe ser hecha en la primera línea del constructor de la subclase. Si no se introduce así, el constructor de la clase derivadada llamará automáticamente al constructor por omisión (sin parámetros) de la superclase.

En el ejemplo siguiente podrá apreciarse esta llamada al constructor de la superclase.

10.2.5 Manejo de objetos de subclase como objetos de superclase en Java

Un objeto de una clase derivada, puede ser manejado como un objeto de su superclase. Sin embargo, un objeto de la clase base no es posible tratarlo como un objeto de clase derivada.

Un objeto de una subclase puede ser asignado a una variable de referencia de su superclase sin necesidad de indicar una conversión explícita mediante enmascaramiento. Cuando si se necesita utilizar enmascaramiento es para asignar de vuelta un objeto que aunque sea de una clase derivada, este referenciado por una variable de clase base. Esta conversión explícita es verificada por la máquina virtual, y si no corresponde el tipo real del objeto, no se podrá hacer la asignación y se generará una excepción en tiempo de ejecución.

Ejemplo:

```
1  Superclase s= new superclase(), aptSuper;
2  Subclase sub= new subclase(), aptSub;
3
4  //válido
5  aptSuper = sub;
6
7  aptSub = (Subclase) aptSuper;
8
9  //inválido
10 aptSub= (Subclase) s;
```

Podemos ver en el ejemplo anterior, que un objeto puede "navegar" en la jerarquía de clases hacia sus superclases, pero no puede ir a una de sus subclases, ni utilizando el enmascaramiento. Esto se hace por seguridad, ya que la subclase seguramente contendrá un mayor número de elementos que una instancia de superclase y estos no podrían ser utilizados porque causarían una inconsistencia.

Por último, es importante señalar que mientras un objeto de clase derivada este referenciado como un objeto de superclase, deberá ser tratado como si el objeto fuera únicamente de la superclase; por lo que no podrá en ese momento tener referencias a atributos o métodos definidos en la clase derivada.

Un código completo se muestra a continuación. Ejemplo:

```
1  // definición de clase point
2  public class Point {
3     protected double x, y; // coordenadas del punto
4
5     // constructor
6     public Point( double a, double b ) {
7             setPoint( a, b );
8     }
9
10    // asigna a x,y las coordenadas del punto
```

```
11     public void setPoint( double a, double b ) {
12        x = a;
13        y = b;
14     }
15
16     // obtiene coordenada x
17     public double getX() {
18             return x;
19     }
20
21     // obtiene coordenada y
22     public double getY() {
23                  return y;
24          }
25
26     // convierte información a cadena
27          public String toString(){
28                  return "[" + x + ", " + y + "]";
29          }
30  }
31  // Definición de clase círculo
32  public class Circle extends Point {  // Hereda de Point
33     protected double radius;
34
35     // constructor sin argumentos
36     public Circle()   {
37        super( 0, 0 );  // llamada a constructor de clase base
38        setRadius( 0 );
39     }
40
41     // Constructor
42     public Circle( double r, double a, double b )   {
43        super( a, b );  // llamada a constructor de clase base
44        setRadius( r );
45     }
46
47     // Asigna radio del círculo
48     public void setRadius( double r )
49        { radius = ( r >= 0.0 ? r : 0.0 ); }
50
51     // Obtiene radio del círculo
52     public double getRadius() { return radius; }
```

```
53
54    // Cálculo área del círculo
55    public double area() { return 3.14159 * radius * radius; }
56
57    // Convierte información en cadena
58    public String toString() {
59       return "Centro = " + "[" + x + ", " + y + "]" +
60              "; Radio = " + radius;
61    }
62 }
63 // Clase de Prueba de las clases Point y Circle
64 public class Prueba {
65
66    public static void main( String argvs[] )  {
67       Point pointRef, p;
68       Circle circleRef, c;
69
70       p = new Point( 3.5, 5.3 );
71       c = new Circle( 2.7, 1.2, 8.9 );
72
73       System.out.println( "Punto p: " + p.toString() );
74       System.out.println( "Circulo c: " + c.toString());
75
76       // Tratamiento del círculo como instancia de punto
77       pointRef = c;   // asigna círculo c a pointRef
78       // en realidad Java lo reconoce dinámicamente como objeto Circle
79       System.out.println( "Circulo c (via pointRef): " + pointRef.toString());
80
81       // Manejar a un círculo como círculo
82       // (obteniéndolo de una referencia de punto)
83       // asigna círculo c a pointref. Se repite la operación por claridad
84       pointRef = c;
85       circleRef = (Circle) pointRef;  // enmascaramiento de superclase a subclase
86       System.out.println( "Circulo c (via circleRef): " + circleRef.toString());
87       System.out.println( "Area de c (via circleRef): " + circleRef.area());
88
89       // intento de referenciar a un objeto point
90       // desde una referencia de Circle (genera una excepcion)
91       circleRef = (Circle) p;
92    }
93 }
```

Listing 25. Ejemplo de objetos de sub clase como de superclase en Java.

10.2.6 Sobreescritura de métodos (Overriding)

Frecuentemente, los métodos heredados no implementan el comportamiento específico requerido por la subclase. Java permite **sobreescribir** (o redefinir) un método de la clase base en la clase derivada. Al invocar el método desde una instancia de la subclase, se ejecutará la nueva versión.

Es una buena práctica utilizar la anotación `@Override` para asegurar que el compilador verifique la firma del método. Sin embargo, aún es posible acceder a la lógica original de la clase base utilizando la referencia `super`.

Ejemplo:

Listing 26. Ejemplo de sobreescritura de métodos y uso de `super`.

10.2.7 Calificador *final*

Es posible que tengamos la necesidad de que cierta parte de una clase no pueda ser modificada en futuras extensiones de la jerarquía de herencia. Para esto es posible utilizar el calificador *final*.

Si un método se especifica en una clase X como *final*:

Sintaxis

```
<acceso> final <tipo> nombreMétodo( <parámetros>)
```

Se esta diciendo que el método no podrá ser redefinido en las subclases de X.

Aunque se omita este calificador, si se trata de un método de clase (estático) o privado, se considera final y no podrá ser redefinido.

Por otro lado, es posible que no queramos dejar la posibilidad de extender una clase, para lo que se utiliza el calificador *final* a nivel de clase:

Sintaxis

```
<acceso> final class nombreClase {

        //definición de la clase

}
```

De esta forma, la clase no permite generar subclases a partir de ella. De hecho, el API de Java incluye muchas clases *final*, por ejemplo la clase java.lang.String no puede ser especializada.

10.2.8 Interfaces en Java

Java únicamente cuenta con manejo de herencia simple, y la razón que se ofrece es que la herencia múltiple presenta algunos problemas de ambigüedad que complica el entendimiento del programa, sin que este tipo de herencia justifique las ventajas obtenidas de su uso.

Sin embargo, es posible que se necesiten recibir características de más de un origen. Java soluciona esto mediante el uso de interfaces, que son una forma para declarar tipos especiales de clase que, aunque con ciertas limitaciones, no ofrecen las complicaciones de la herencia múltiple.

Una interfaz tiene un formato muy similar a una clase, sus principales características:

- Una interfaz proporciona los nombres de los métodos (método abstracto), pero no sus implementaciones[2].
- Los métodos son públicos por omisión.
- Una clase puede implementar varias interfaces, aunque solo pueda heredar una clase.
- No es posible crear instancias de una interfaz.
- La clase que implementa la interfaz debe escribir el código de todos los métodos, de otra forma no se podrá generar instancias de esa clase.

El formato general para la declaración de una interfaz es el siguiente:

Sintaxis

```
[public] interface <nombreInterfaz> {
        //descripción de miembros
        //los métodos no incluyen código:
        <acceso> <tipo> <nombreMetodo> ( <parámetros> ) ;
}
```

[2]En esta caso si se considera la declaración de prototipos.

El cuerpo de la interfaz generalmente es una lista de prototipos de métodos, pero puede contener atributos si se requiere[3].

Una clase implementa una interfaz a través de la palabra reservada *implements* después de la especificación de la herencia (si la hubiera) :

Sintaxis

```
class <SubClase> extends <Superclase> implements   <nombreInterfaz> {
        //definición de la clase
        //debe incluirse la definición de los métodos de la interfaz
        //con la implementación del código de dichos métodos.
}
```

Además, una interfaz puede ser extendida de la misma forma que una clase, aprovechando las interfaces previamente definidas, mediante el uso de la cláusula *extends*.

Sintaxis

```
[public] interface <nombreInterfaz> extends <InterfazBase> {

        //descripción de miembros

}
```

De forma distinta a la jerarquía de clases, donde se tiene una jerarquía lineal que parte siempre de una clase simple *Object*, una clase soporta herencia múltiple de interfaces, resultando en una jerarquía con múltiples raíces de diferentes interfaces.

Ejemplo:

```
1 //interfaz
2 interface IStack {
3     void push(Object item);
```

[3]El parámetro debe incluir el nombre, el cual no es obligatorio que coincida en la implementación.

```
4       Object pop();
5   }
6
7   //clase implementa la interfaz
8   class StackImpl implements IStack {
9       protected Object[] stackArray;
10      protected int tos;
11
12      public StackImpl(int capacity) {
13          stackArray = new Object[capacity];
14          tos = -1;
15      }
16
17          //implementa el método definido en la interfaz
18      public void push(Object item)
19          { stackArray[++tos] = item; }
20
21          //implementa el método definido en la interfaz
22      public Object pop() {
23          Object objRef = stackArray[tos];
24          stackArray[tos] = null;
25          tos--;
26          return objRef;
27      }
28
29      public Object peek() { return stackArray[tos];}
30  }
31
32  // extendiendo una interfaz
33  interface ISafeStack extends IStack {
34      boolean isEmpty();
35      boolean isFull();
36  }
37
38
39  //esta clase hereda la implementación de la pila StackImpl
40  // e implementa la nueva interfaz extendida ISafeStack
41  class SafeStackImpl extends StackImpl implements ISafeStack {
42
43      public SafeStackImpl(int capacity) { super(capacity); }
44
45      //implementa los métodos de la interfaz
```

```
46      public boolean isEmpty() { return tos < 0; }
47      public boolean isFull() { return tos >= stackArray.length;
48           }
49  }
50
51  public class StackUser {
52
53      public static void main(String args[]) {
54          SafeStackImpl safeStackRef = new SafeStackImpl(10);
55          StackImpl stackRef = safeStackRef;
56          ISafeStack isafeStackRef = safeStackRef;
57          IStack istackRef = safeStackRef;
58          Object objRef = safeStackRef;
59
60          safeStackRef.push("Dolar");
61          stackRef.push("Peso");
62
63                  // tipo de dato simple es convertido a Integer:
64          stackRef.push(1);
65          System.out.println(stackRef.peek().getClass());
66
67          System.out.println(isafeStackRef.pop());
68          System.out.println(istackRef.pop());
69
70          System.out.println(objRef.getClass());
71      }
72  }
```

Listing 27. Ejemplo de interfaz en Java.

Por otro lado, una interfaz también puede ser utilizada para definir nuevos tipos. Una interfaz así o una clase que implementa a una interfaz de este estilo es conocida como **Supertipo**.

Es importante resaltar tres diferencias en las relaciones de herencia y como esta funciona entre clases e interfaces:

1. **Implementación lineal de jerarquía de herencia entre clases**: una clase extiende a otra clase.

2. **Jerarquía de herencia múltiple entre interfaces**: una interfaz extiende otras interfaces.

3. **Jerarquía de herencia múltiple entre interfaces y clases**: una clase implementa interfaces.

Interfaces en Java con implementación predeterminada de métodos

A partir de Java 8 permite el uso de métodos implementados en interfaces. Estos pueden ser métodos estáticos o un tipo especial de método llamado *Default*[4].

Sintaxis

```
[public] interface <nombreInterfaz> {
        //implementación predeterminada de método
        <acceso> default <tipo> <nombreMetodo> ( <parámetros> ) {
            //código del método
    }
}
```

Ahora, las clases que implementan la interfaz no tienen que proporcionar su propia implementación del método o métodos. Si no lo hacen, se utilizará la implementación predeterminada.

Ambigüedad con implementación de métodos en interfaces

Si una clase hereda de una interfaz que tiene un método por defecto y también hereda de una clase base que tiene un método con la misma firma, se utilizará la implementación de la clase base, y no se generará una ambigüedad.

Pero si de dos interfaces distintas se hereda un método implementado con la misma firma, si existe ambigüedad. Para evitarla, la clase que implementa a las interfaces tiene queproporcionar su propia implementación del método:

Ejemplo: Ejemplo:

```
1
2  interface InterfazA {
3      default void metodo() {
4          System.out.println("InterfazA");
5      }
6  }
7
8  interface InterfazB {
```

[4] https://www.techempower.com/blog/2013/03/26/everything-about-java-8/, https://stackoverflow.com/questions/18286235/what-is-the-default-implementation-of-method-defined-in-an-interface

```
    default void metodo() {
        System.out.println("InterfazB");
    }
}

// Ambigüedad si descomentas esta línea:
// class MiClase implements InterfazA, InterfazB {}

class MiClase implements InterfazA, InterfazB {
    // Si proporcionas tu propia implementación, no hay ambigüedad
    @Override
    public void metodo() {
        System.out.println("Implementación en MiClase");
    }
}

public class Main {
    public static void main(String[] args) {
        MiClase instancia = new MiClase();
        instancia.metodo(); // Imprime "Implementación en MiClase"
    }
}
```

Listing 28. Ejemplo de interfaz con potencial ambigüedad en métodos predefinidos en Java.

Y ¿atributos en las interfaces?

Hasta Java 7, las interfaces solo podían contener constantes (variables estáticas finales) como atributos. Estos atributos se consideraban **automáticamente** como *public*, *static*, y *final*. Un ejemplo sería:

```
public interface MiInterfaz {
    int MI_CONSTANTE = 42; // Atributo (public, static, final)
}
```

Capítulo 11

Asociaciones entre clases

11.1 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).

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.

[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.

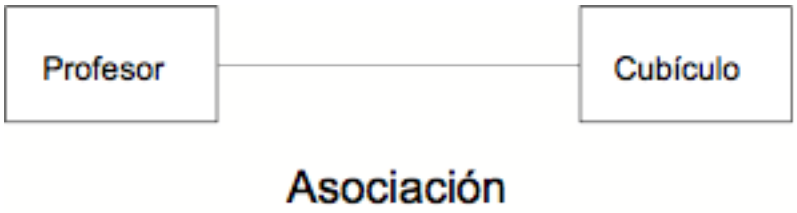


Fig. 11.1. Ejemplo de asociación en UML

En el caso de las asociaciones se asumirá que cada objeto puede seguir existiendo de manera independiente, a menos que haya sido creado por el objeto de la clase asociada, en cuyo caso deberá ser eliminado por el destructor del objeto que la creó. Es decir:

Explicación
Si el objeto **A** crea al objeto **B**, es responsabilidad de **A** eliminar a la instancia **B** antes de que **A** sea eliminada. En caso contrario, si **B** es independiente de la instancia **A**, **A** debería enviar un mensaje al objeto **B** para que asigne *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.

11.2 Asociaciones reflexivas

Es posible tener un tipo de asociación conocida como asociación **reflexiva**.

Concepto
Si una clase mantiene una asociación consigo misma se dice que es una **asociación reflexiva**.

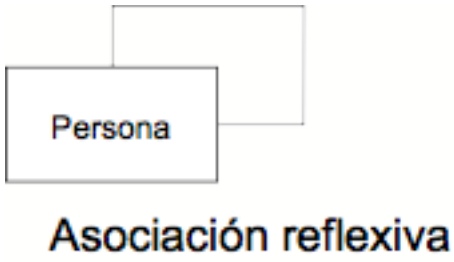


Fig. 11.2. Asociación reflexiva

Ejemplo: *Persona* puede tener asociaciones entre sí, si lo que nos interesa es representar a las personas que guardan una relación entre sí, por ejemplo si son parientes. Es decir, un objeto mantiene una asociación con otro objeto de la misma clase.

En términos de implementación significa que la clase tiene una referencia a si misma. De nuevo podemos poner de ejemplo a la clase Nodo en una lista ligada.

11.3 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.

11.3.1 Tipos de asociaciones según su multiplicidad

"uno a uno": donde dos objetos se asocian de forma exclusiva, uno con el otro. Ejemplo: Uno: Un alumno tiene una boleta de calificaciones. Uno: Una boleta de calificaciones pertenece a un alumno.

"uno a muchos": donde uno de los objetos puede estar asociado con muchos otros objetos. Ejemplo: Uno: un libro solo puede estar prestado a un alumno. Muchos: Un usuario de la biblioteca puede tener muchos libros prestados.

"muchos a muchos": donde cada objeto de cada clase puede estar asociado con muchos otros objetos. Ejemplo: Muchos: Un libro puede tener varios autores. Muchos: Un autor puede tener varios libros.

Podemos apreciar en un diagrama las diversas multiplicidades:

Finalmente, es importante señalar que el control de las asociaciones no se encuentra en general apoyado por los lenguajes de programación, a pesar de ser una necesidad natural en el modelado orientado a objetos, por lo que toda la responsabilidad recae sobre el programador.

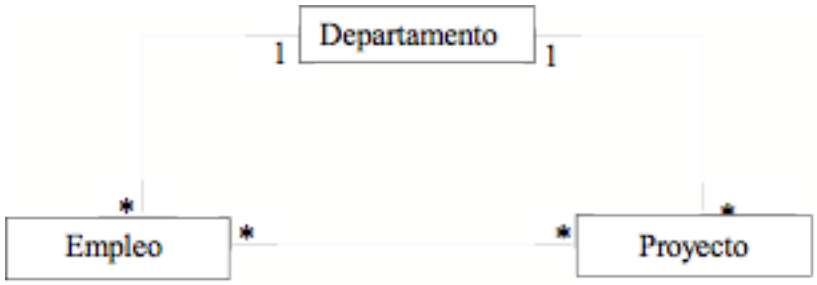


Fig. 11.3. Ejemplo de multiplicidad

11.4 Asociaciones en Java

Como en Java el manejo de objetos es mediante referencias, la implementación de la asociación se simplifica en la medida que la sintaxis de Java es más simple.

Ejemplo: un código que guarda una asociación bidireccional entre clases A y B.

```
1  class A{
2          //lista de atributos
3
4          B         pB;
5  }
6
7  class B{
8          //lista de atributos
9          A         pA;
10 }
```

En el ejemplo anterior se presenta una relación bidireccional, por lo que cada clase tiene su respectiva referencia a la clase contraria de la relación. Además, deben proporcionarse métodos de acceso a la clase relacionada por medio de la referencia.

Una asociación unidireccional del ejemplo anterior sería más simple. Veamos el código si se requiere únicamente una relación de A a B.

Ejemplo:

```
1  class A{
2          //lista de atributos
3          B         pB;
4  }
5
```

```
6  class B{
7          //lista de atributos
8  }
```

Recordar

Si el objeto **A** crea al objeto **B**, es responsabilidad de **A** eliminar a la instancia **B** antes de que **A** sea eliminada. En caso contrario, si **B** es independiente de la instancia **A**, **A** debería enviar un mensaje al objeto **B** para que asigne *null* al apuntador de **B** o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida.

En Java, ya que cuenta con un recolector de basura, la importancia radicaría en asegurarnos de no mantener enlaces a objetos que ya no son necesarios.

11.4.1 Multiplicidad de una asociación en Java

La forma de implementar en Java este tipo de relaciones puede variar, pero la más común es por medio de referencias a objetos. Suponiendo que tenemos relaciones bidireccionales:

- **"uno a uno"**. Una referencia de cada lado de la relación, como se ha visto anteriormente.
- **"uno a muchos"**. Una referencia de un lado y un arreglo de referencias a objetos del otro lado.

```
1  class A{
2          ...
3          B pB;
4  }
5  class B{
6          A p[];
7  }
```

Al igual que en C++, es posible manejar una clase independiente que agrupe a pares de direcciones en un objeto independiente de la clase[2]. Por ejemplo, en una estructura de lista.

[2]Ver figura en tema correspondiente de C++

- **"muchos a muchos"**. Normalmente se utiliza un objeto u objetos independientes que mantiene las relaciones entre los objetos, de manera similar a la solución descrita en el punto anterior.

Ejemplo: Se muestra un código simplificado para manejo de asociaciones.

```
1  //clase Libro
2  class Libro {
3          private String nombreLibro;
4          public Alumno pAlumno;
5
6          public Libro(){
7                  //al momento de crearse la instancia, no existe
8                  // relación con ningún Alumno
9                  pAlumno=null;
10         }
11
12         protected void finalize(){
13                 //si es diferente de null, el libro está
14                 //asignado a algún Alumno
15                 if(pAlumno!=null)
16                         //busca la referencia de Alumno a
17                         //Libro para ponerla en null
18                         for(int i=0; i<5; i++)
19                                 if (pAlumno.pLibrosPres[i]==this)
20                                         pAlumno.pLibrosPres[i]=null;
21         }
22 }
23
24 //clase Alumno
25 class Alumno {
26         public Libro pLibrosPres[];
27         public Alumno(){
28                 int i;
29                 //se asume una multiplicidad de 5
30                 pLibrosPres = new Libro[5]
31                 for(i=0; i<5; i++)
32                         pLibrosPres[i]=null;
33         }
34
35         protected void finalize(){
36
```

```
37                  //pone en null todas las asociaciones de los Libros
38                  // a su instancia de Alumno que se elimina
39                  for(int i=0; i<5; i++)
40                          if(pLibrosPres[i]!=null)
41                                  pLibrosPres[i].pAlumno=null;
42          }
43  }
```

Este es un ejemplo parcial de cómo se soluciona el manejo de asociaciones entre clases, ya que además se deben de agregar métodos para establecer y eliminar la asociación, en ambas clases si es una asociación bidireccional, o en una clase únicamente si se trata de una asociación unidireccional. Esos deben de ser los únicos métodos que tengan el control sobre los atributos que mantienen la asociación y no deberían ser manejados directamente, por lo que no deben ser públicos como aquí se presentaron.

Una definición más completa - sin implementación - de la clase *Libro* se aprecia a continuación:

```
1   class Libro {
2           private String nombreLibro;
3           private String clave;
4           public Alumno pAlumno;
5
6           public Libro(){         }
7
8           public Libro( Alumno pAlumno){          }
9
10          public String getNombreLibro(){          }
11
12          public void setNombreLibro(String n){         }
13
14          public String getClave(){          }
15
16          public void setClave(String cve){          }
17
18          public boolean setAsociacion(Alumno pAlumno){          }
19
20          public boolean unsetAsociacion(){          }
21
22          public Alumno getAlumno(){         }
23
```

```
24          protected finalize {}
25  }
```

Este sería un estilo más apropiado para el desarrollo de asociaciones, aunque existen otros más elaborados.

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 12

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.

12.1 Objetos compuestos en Java

En Java, puede no existir mucha diferencia entre la implementación de la asociación y la agregación, debido a que en Java los objetos siempre son manejados por referencias, pero el concepto se debe tener en cuenta para su manejo, además de ser relevante a nivel de diseño de software.

Recordemos que en general hay dos opciones para el manejo de la agregación:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.

2. Cuando el agregado es destruido también sus componentes se destruyen.

Al igual que en C++, vamos a considerar la segunda opción, por ser más fácil de implementar y es la acción natural de los objetos que se encuentran embebidos como un atributo más una clase.

Ejemplo:

```
1  class Nombre {
2          private String paterno;
3          private String materno;
4          private String nom;
5
6          public        set(String pat, String mat, String n) {
7                  ...
8          }
9                  ...
10 }
11
12 class Persona {
13         private int edad;
14         private Nombre nombrePersona;
15         ...
16 }
```

A diferencia de lo que sucede en C++, los atributos compuestos no tienen memoria asignada, es decir, los objetos compuestos no han sido realmente creados en el momento en que se crea el objeto componente. Es responsabilidad del constructor del objeto componente inicializar los objetos miembros o compuestos, si es que así se requiere.

Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos set correspondientes a la modificación de los atributos de los objetos componentes, esto claro está, después de asignarle la memoria a los objetos componentes.

2. Llamar a algún constructor especializado del objeto componente en el momento de crearlo.

Ejemplo:

```
1  //Programa Persona
2  class Nombre {
3          private String nombre,
4                  paterno,
5                  materno;
6          public Nombre(String n, String p, String m){
7                          nombre= new String(n);
8                          paterno= new String(p);
9                          materno= new String(m);
10         }
11 }
12
13 public class Persona{
14         private Nombre miNombre;
15         private int edad;
16         public Persona(String n, String p, String m) {
17                 miNombre= new Nombre(n, p, m);
18                 edad=0;
19         }
20
21         public static void main(String args[]) {
22                 Persona per1;
23                 per1= new Persona("uno", "dos", "tres");
24                 Persona per2= new Persona("mi nombre", "mi apellido",
25                   "otro apellido");
26         }
27 }
```

Listing 29. Ejemplo de composición en Java.

Pero también es posible que un objeto sea un agregado recursivo, es decir, tener como parte de su componente un objeto de su misma clase. Considerar por ejemplo un directorio de archivos, donde cada directorio puede contener, además de archivos, a otros directorios[1].

[1]Lo importante aquí es considerar en que solo existe la **posibilidad** de contener un objeto de si

mismo. Si esto fuera una condición obligatoria y no opcional, estaríamos definiendo un **objeto infinito**. Este problema se ve reflejado en lenguajes como C++, donde la forma más simple de implementar la agregación es definiendo un objeto al cual se le asigna espacio en tiempo de compilación, generando entonces el problema de que cada objeto debe reservar memoria para sus componentes, por lo que el compilador no permite que de esta manera se autocontenga. En Java esto no generaría problema porque implícitamente todos los atributos que no son datos simples requieren de una asignación de memoria dinámica.

Capítulo 13

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 14

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.

14.1 Polimorfismo y clases abstractas Java

El polimorfismo es implementado en Java a través de clases derivadas y clases **abstractas**.

Concepto
Recordar: El **polimorfismo** se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método.

Al hacer una solicitud de un método, a través de una variable de referencia a clase base para usar un método, Java determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Los métodos en Java - a diferencia de C++ - tienen este comportamiento por default, debido a que cuando un método es accedido por una referencia a una clase base, y esta mantiene una referencia a un objeto de una clase derivada, el programa determina **en tiempo de ejecución** a que método llamar, de acuerdo al tipo de objeto al que se apunta.

Esto como ya se ha visto, se conoce como **ligadura tardía** y permite otro nivel de reutilización de código, resaltado por la simplificación con respecto a C++ de no tener que declarar al método como virtual.

Ejemplo:

```
1  //ejemplo Prueba
2  class base {
3          public void quien(){
4                  System.out.println("base");
5          }
6  }
7
8  class primera extends base {
9          public void quien(){
10                 System.out.println("primera");
11         }
12 }
13
14 class segunda extends base {
15         public void quien(){
16                 System.out.println("segunda");
17         }
```

```
18 }
19
20 class tercera extends base { }
21
22 class cuarta extends base {
23         /*public int quien(){   No se vale con un tipo de dato diferente
24                 System.out.println("cuarta");
25                  return 1;
26          }*/
27 }
28
29 public class Prueba {
30         public static void main(String args[]) {
31                 base objBase= new base(), pBase;
32                 primera obj1= new primera();
33                 segunda obj2= new segunda();
34                 tercera obj3= new tercera();
35                 cuarta  obj4= new cuarta();
36
37                 pBase=objBase;
38                 pBase.quien();
39
40                 pBase=obj1;
41                 pBase.quien();
42
43                 pBase=obj2;
44                 pBase.quien();
45
46                 pBase=obj3;
47                 pBase.quien();
48
49                 pBase=obj4;
50                 pBase.quien();
51         }
52 }
```

Listing 30. Ejemplo de polimorfismo en Java.

Como se aprecia en el ejemplo anterior, en caso de que el método no sea redefinido, se ejecuta el método de la clase base.

Es importante señalar que – al igual que en C++- los métodos que sean redefinidos en clases derivadas, deben tener además de la misma firma que método base, el mismo tipo de retorno. Si se declara en una clase derivada un método con otro

tipo de dato como retorno, se generará un error en tiempo de compilación.

14.1.1 Clase abstracta y clase concreta en Java

Concepto

Recordar: Una **clase base abstracta**, es aquella que es definida para especificar características generales que van a ser aprovechadas por sus clases derivadas, pero no se necesita instanciar a dicha superclase.

Sintaxis

```
abstract class ClaseAbstracta {
        //código de la clase
}
```

Además, existe la posibilidad de contar con métodos abstractos:

Concepto

Un **método abstracto** lleva la palabra reservada *abstract* y contiene sólo el nombre y su firma. No necesita implementarse, ya que esto es tarea de las subclases.

Si una clase contiene al menos un método abstracto, toda la clase es considerada abstracta y es conveniente, por claridad, declararla como tal. Es posible claro, declarar a una clase como abstracta sin que tenga métodos abstractos.

Ejemplo básico para un método abstracto:

```
1 abstract class ClaseAbstracta {
2
3         public abstract void noTengoCodigo( int x);
4
5 }
```

Si se crea una subclase de una clase que contiene un método abstracto, deberá de especificarse el código de ese método; de lo contrario, el método seguirá siendo abstracto y por consecuencia también lo será la subclase[1].

[1]En C++, una clase se hace abstracta al declarar al menos uno de los métodos virtuales como puro.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener referencias a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante las referencias a la clase abstracta.

El uso de clases abstractas **fortalece** al polimorfismo, al poder partir de clases definidas en lo general, sin implementación de código, pero pudiendo ser agrupadas todas mediante variables de referencia a las clases base.

Ejemplos de clases abstractas y polimorfismo:

Programa de cálculo de salario

```
1  // Clase base abstracta Employee
2  public abstract class Employee {
3          private String firstName;
4          private String lastName;
5
6             // Constructor
7             public Employee( String first, String last )  {
8        firstName = new String ( first );
9        lastName = new String( last );
10            }
11
12            public String getFirstName()  {
13                    return new String( firstName );
14         }
15
16            public String getLastName()   {
17                    return new String( lastName );
18         }
19
20            // el metodo abstracto debe de ser implementado por cada
21            // clase derivada de Employee para poder ser
22            // instanciadas las subclases
23            public abstract double earnings();
24 }
25
26 // Clase Boss class derivada de Employee
27 public final class Boss extends Employee {
28         private double weeklySalary;
29
30 public Boss( String first, String last, double s) {
31                 super( first, last );  // llamada al constructor de clase base
32                 setWeeklySalary( s );
33            }
```

```
34
35          public void setWeeklySalary( double s ){
36                  weeklySalary = ( s > 0 ? s : 0 );
37          }
38
39          // obtiene pago del jefe
40          public double earnings() {
41                  return weeklySalary;
42          }
43
44          public String toString() {
45                  return "Jefe: " + getFirstName() + ' ' +
46               getLastName();
47          }
48  }
49
50  // Clase PieceWorker derivada de Employee
51  public final class PieceWorker extends Employee {
52          private double wagePerPiece; // pago por pieza
53          private int quantity;        // piezas por semana
54
55          public PieceWorker( String first, String last,
56                          double w, int q )    {
57                  super( first, last );
58                  setWage( w );
59                  setQuantity( q );
60          }
61
62          public void setWage( double w )    {
63                  wagePerPiece = ( w > 0 ? w : 0 );
64          }
65
66          public void setQuantity( int q )    {
67                  quantity = ( q > 0 ? q : 0 );
68          }
69
70          public double earnings()   {
71                  return quantity * wagePerPiece;
72          }
73
74          public String toString()   {
75                  return "Trabajador por pieza: " +
```

```
76              getFirstName() + ' ' + getLastName();
77          }
78  }
79
80  // Clase HourlyWorker derivada de Employee
81  public final class HourlyWorker extends Employee {
82          private double wage;   // pago por hora
83          private double hours;  // horas trabajadas por semana
84
85          public HourlyWorker( String first, String last,
86                               double w, double h )    {
87                  super( first, last );
88                  setWage( w );
89                  setHours( h );
90          }
91
92          public void setWage( double w )    {
93                  wage = ( w > 0 ? w : 0 );
94          }
95
96          public void setHours( double h )    {
97                  hours = ( h >= 0 && h < 168 ? h : 0 );
98          }
99
100         public double earnings()    {
101                 return wage * hours;
102         }
103
104         public String toString()    {
105                 return "Trabajador por hora: " +
106              getFirstName() + ' ' + getLastName();
107         }
108 }
109
110 // Clase CommissionWorker derivada de Employee
111 public final class CommissionWorker extends Employee {
112         private double salary;      // salario base por semana
113         private double commission;  // monto por producto vendido
114         private int quantity;       // total de productos vendidos por semana
115
116         public CommissionWorker( String first, String last,
117                                  double s, double c, int q)    {
```

```
118                 super( first, last );
119                 setSalary( s );
120                 setCommission( c );
121                 setQuantity( q );
122         }
123
124         public void setSalary( double s )   {
125                 salary = ( s > 0 ? s : 0 );
126         }
127
128         public void setCommission( double c )    {
129                 commission = ( c > 0 ? c : 0 );
130         }
131
132         public void setQuantity( int q )   {
133                 quantity = ( q > 0 ? q : 0 );
134         }
135
136         public double earnings()   {
137                 return salary + commission * quantity;
138         }
139
140         public String toString()   {
141                 return "Trabajador por Comision : " +
142              getFirstName() + ' ' + getLastName();
143         }
144 }
145
146 // Programa de ejemplo Polimorfismo
147 public class Polimorfismo {
148         public static void main( String rgs[] ) {
149                 Employee ref;  // referencia de clase base
150                 Boss b;
151                 CommissionWorker c;
152                 PieceWorker p;
153                 HourlyWorker h;
154                 b = new Boss( "Alan", "Turing", 800.00 );
155                 c = new CommissionWorker( "Ada", "Lovelace",
156                                 400.0, 3.0, 150);
157                 p = new PieceWorker( "Grace", "Hopper", 2.5, 200 );
158                 h = new HourlyWorker( "James", "Gosling", 13.75, 40 );
159
```

```
160                 ref = b;  // referencia de superclase a objeto de subclase
161                 System.out.println( ref.toString() + " ganó $" +
162                     ref.earnings() );
163                 System.out.println( b.toString() + " ganó $" +
164                     b.earnings() );
165
166                 ref = c;  // referencia de superclase a objeto de subclase
167                 System.out.println( ref.toString() + " ganó $" +
168                     ref.earnings() );
169                 System.out.println( c.toString() + " ganó $" +
170                     c.earnings() );
171
172                 ref = p;  // referencia de superclase a objeto de subclase
173                 System.out.println( ref.toString() + " ganó $" +
174                     ref.earnings() );
175                 System.out.println( p.toString() + " ganó $" +
176                     p.earnings() );
177
178                 ref = h;  // referencia de superclase a objeto de subclase
179                 System.out.println( ref.toString() + " ganó $" +
180                     ref.earnings() );
181                 System.out.println( h.toString() + " ganó $" +
182                     h.earnings() );
183         }
184 }
```

Listing 31. Ejemplo de clase abstracta y polimorfismo en Java. Programa de cálculo de salario

Ejemplo:**Programa de figuras geométricas con una clase abstracta Shape (Forma)**

```
1  // Definicion de clase base abstracta Shape
2  public abstract class Shape {
3
4          public double area() {
5                  return 0.0;
6          }
7
8          public double volume() {
9                  return 0.0;
10         }
11
```

```
12          public abstract String getName();
13  }
14
15  // Definicion de clase Point
16  public class Point extends Shape {
17          protected double x, y; // coordenadas del punto
18
19          public Point( double a, double b ) { setPoint( a, b ); }
20
21          public void setPoint( double a, double b )    {
22                  x = a;
23                  y = b;
24          }
25
26          public double getX() { return x; }
27
28          public double getY() { return y; }
29
30          public String toString()
31        { return "[" + x + ", " + y + "]"; }
32
33          public String getName() {
34                  return "Punto";
35          }
36  }
37
38  // Definicion de clase Circle
39  public class Circle extends Point {  // hereda de Point
40          protected double radius;
41
42          public Circle()    {
43                  super( 0, 0 );
44                  setRadius( 0 );
45          }
46
47          public Circle( double r, double a, double b )    {
48                  super( a, b );
49                  setRadius( r );
50          }
51
52          public void setRadius( double r )
53        { radius = ( r >= 0 ? r : 0 ); }
```

```
54
55          public double getRadius() { return radius; }
56
57          public double area() { return 3.14159 * radius * radius; }
58
59          public String toString()
60        { return "Centro = " + super.toString() +
61                 "; Radio = " + radius; }
62
63          public String getName() {
64                   return "Circulo";
65          }
66  }
67
68  // Definicion de clase Cylinder
69  public class Cylinder extends Circle {
70          protected double height;  // altura del cilindro
71
72          public Cylinder( double h, double r, double a, double b )          {
73                   super( r, a, b );
74                   setHeight( h );
75          }
76
77          public void setHeight( double h ){
78                   height = ( h >= 0 ? h : 0 );
79          }
80
81          public double getHeight() {
82                   return height;
83          }
84
85          public double area()    {
86                   return 2 * super.area() +
87                            2 * 3.14159 * radius * height;
88          }
89
90          public double volume() {
91                   return super.area() * height;
92          }
93
94          public String toString(){
95                   return super.toString() + "; Altura = " + height;
```

```
96          }
97
98          public String getName() {
99                  return "Cilindro";
100         }
101 }
102
103 // Codigo de prueba
104 public class Polimorfismo02 {
105
106         public static void main (String args []) {
107                 Point point;
108                 Circle circle;
109                 Cylinder cylinder;
110                 Shape arrayOfShapes[];
111
112                 point = new Point( 7, 11 );
113                 circle = new Circle( 3.5, 22, 8 );
114                 cylinder = new Cylinder( 10, 3.3, 10, 10 );
115
116                 arrayOfShapes = new Shape[ 3 ];
117
118                 // asigno las referencias de los objetos de subclase
119                 // a un arreglo de superclase
120                 arrayOfShapes[ 0 ] = point;
121                 arrayOfShapes[ 1 ] = circle;
122                 arrayOfShapes[ 2 ] = cylinder;
123
124                 System.out.println( point.getName() + ": " + point.toString());
125
126                 System.out.println( circle.getName() + ": " + circle.toString());
127
128                 System.out.println( cylinder.getName() + ": " + cylinder.toString());
129
130                 for ( int i = 0; i < 3; i++ ) {
131                         System.out.println( arrayOfShapes[ i ].getName() +
132                                         ": " + arrayOfShapes[ i ].toString());
133                         System.out.println( "Area = " + arrayOfShapes[ i ].area() );
134                         System.out.println( "Volume = " + arrayOfShapes[ i ].volume() )
135                 }
136         }
```

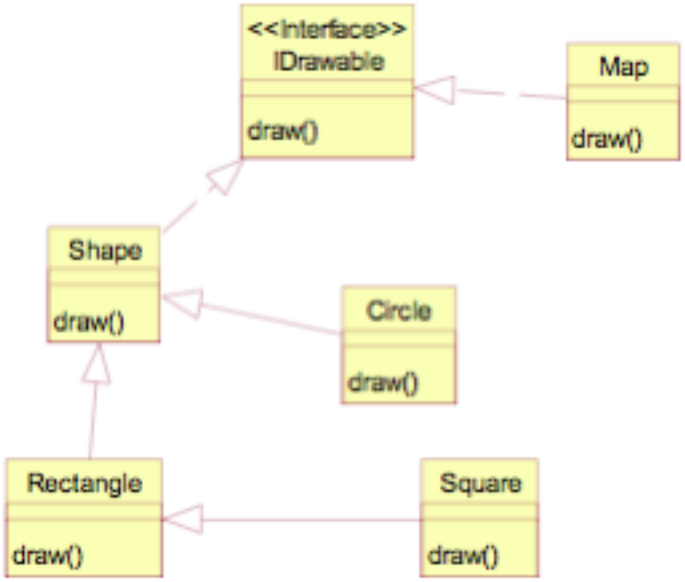


Fig. 14.1. Ejemplo de polimorfismo con interfaces en Java

```
137 }
```

Listing 32. Ejemplo de polimorfismo en Java. Programa de figuras geométricas con una clase abstracta Shape (Forma)

Clase abstracta o interfaz

14.1.2 Ejemplo de Polimorfismo con una Interfaz en Java

Los programas anteriores estaban basados en clases y clases abstractas. Sin embargo, también es posible tener variables de referencia a interfaces, a través de las cuales se implemente el polimorfismo. El siguiente programa muestra otra estructura clásica de clases "gráficas", todas contienen su propia implementación de *draw*(), y son organizadas en dos arreglos de ejemplo: uno de la clase principal, y el segundo del tipo de la interfaz.

Ejemplo:

```
1 //programa Polimorfismo
2 interface IDrawable {
3     void draw();
4 }
5
6 class Shape implements IDrawable {
```

```
7       public void draw() { System.out.println("Dibujando Figura."); }
8   }
9
10  class Circle extends Shape {
11      public void draw() { System.out.println("Dibujando Circulo."); }
12  }
13
14  class Rectangle extends Shape {
15      public void draw() { System.out.println("Dibujando Rectangulo."); }
16  }
17
18  class Square extends Rectangle {
19      public void draw() { System.out.println("Dibujando cuadrado."); }
20  }
21
22  class Map implements IDrawable {
23      public void draw() { System.out.println("Dibujando mapa."); }
24  }
25
26  public class Polimorfismo03 {
27      public static void main(String args[]) {
28          Shape[] shapes = {new Circle(), new Rectangle(), new Square()};
29          IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};
30
31          System.out.println("Dibujando figuras:");
32          for (int i = 0; i < shapes.length; i++)
33              shapes[i].draw();
34
35          System.out.println("Dibujando elementos dibujables:");
36          for (int i = 0; i < drawables.length; i++)
37              drawables[i].draw();
38      }
39  }
```

Listing 33. Ejemplo de polimorfismo con interfaces en Java.

Capítulo 15

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**.

15.1 Clases Genéricas en Java

Java 1.5 introdujo finalmente el uso de clases genéricas (*generics*)[**?**]. El uso de clases genéricas es una característica poderosa usada en otros lenguajes, siendo C++ el ejemplo más conocido que soporta programación genérica mediante el uso de plantillas o *templates*.

Sintaxis
`class NombreClase <Lista de parámetros de tipos> { ... }`

Ejemplo:

```
1  class Pair<T, U> {
2    private final T first;
3    private final U second;
4    public Pair(T first, U second) { this.first=first; this.second=second; }
5    public T getFirst() { return first; }
6    public U getSecond() { return second; }
7  }
8
9  public class PairExample {
10         public static void main(String[] args) {
11
12                 Pair<String, Integer> pair = new Pair<String, Integer>("one",2);
13
14                 // no acepta tipos de datos básicos o primitivos
15                 //Pair<String, int> pair2 = new Pair<String, Integer>("one",2);
16
17                 // siguiente linea generaría un warning de seguridad de tipos
18                 //Pair<String, Integer> pair3 = new Pair("one",2);
19
20                 System.out.println("Obtén primer elemento:" + pair.getFirst());
21                 System.out.println("Obtén segundo elemento:" + pair.getSecond());
22         }
23 }
```

Listing 34. Ejemplo de clases genéricas en Java.

Es también posible parametrizar interfaces, como se muestra a continuación.

Sintaxis
`interface NombreInterfaz <Lista de parámetros de tipos> { ... }`

Ejemplo:

```
interface IPair<T, U>{
        public T getFirst();
        public U getSecond();
}

class Pair<T, U> implements IPair<T, U>{
  private final T first;
  private final U second;
  public Pair(T first, U second) { this.first=first; this.second=second; }
  public T getFirst() { return first; }
  public U getSecond() { return second; }
}

public class PairExample {
        public static void main(String[] args) {

                IPair<String, Integer> ipair = new Pair<String, Integer>("one",2);

                System.out.println("Obtén primer elemento:"+ipair.getFirst());
                System.out.println("Obtén segundo elemento:"+ipair.getSecond());
        }

}
```

Listing 35. Ejemplo clases e interfaces genéricas en Java.

Un requerimiento para el uso tipos genéricos en Java es que no pueden usarse tipos de datos primitivos, porque los tipos primitivos o básicos no son subclases de Object[**?**]. Por lo que sería ilegal por ejemplo querer instanciar $Pair < int, String >$. La ventaja es que el uso de la clase *Object* significa que solo un archivo de clase (*.class*) necesita ser generado por cada clase genérica[**?**].

Restriciones de las clases genéricas, ver[1].

15.2 Biblioteca de Clases Genéricas en Java

Al igual que C++ con la STL, Java tiene un conjunto de clases genéricas predefinidas. Su uso, de manera similar que con las clases genéricas definidas por el programador, no está permitido para tipos primitivos, por lo que solo objetos podrán ser contenidos. Las principales clases genéricas en Java son, como en la STL, clases

[1] Java generics

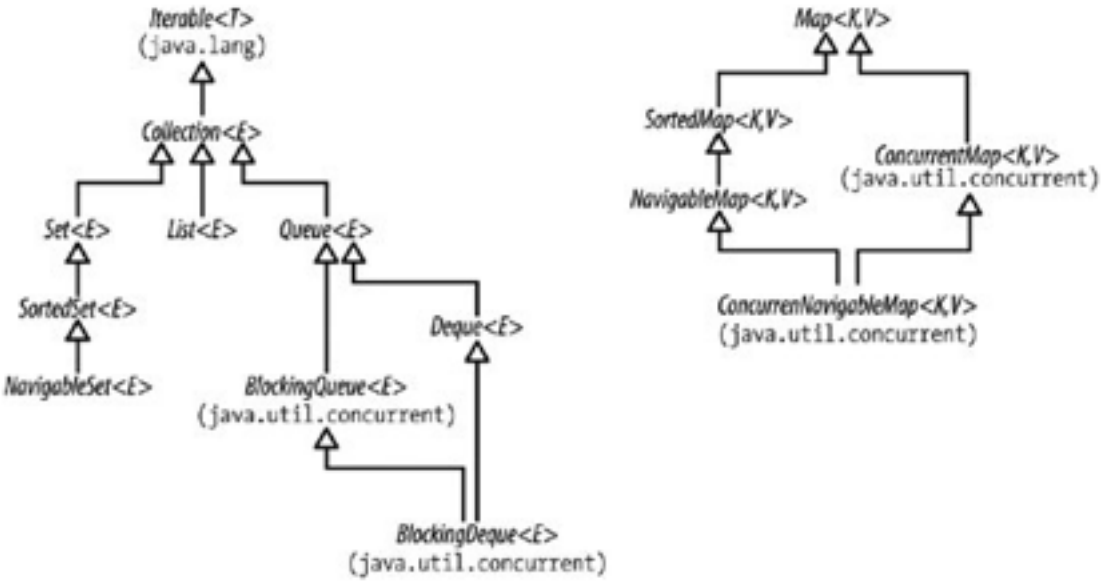


Fig. 15.1. Biblioteca genérica en Java

contenedoras o colecciones[2]. El *Java Collections Framework* (JCF) es un conjunto de interfaces y clases definidos en los paquetes *java.util* y *java.util.concurrent*.

Las interfaces del JCF son:

- *Collection*. Contiene la funcionalidad básica requerida en casi cualquier colección de objetos (con excepción de *Map*)
- *Set*. Es una colección sin duplicados, donde el orden es no significante. Sin embargo contiene un método que devuelve el conjunto ordenado (*SortedSet*).
- *Queue*. Define el comportamiento básico de una estructura de cola.
- *List*. Es una colección donde el orden es significativo, permitiendo además valores duplicados.
- *Map*. Define una colección donde un valor clave es asociado para almacenar y recuperar elementos.

La siguiente figura muestra las principales interfaces de la JCF[**?**]:

Los iteradores son objetos que te permiten recorrer una colección de objetos, obteniendo o removiendo elementos. Un objeto iterador implementa la interfaz

[2]Las colecciones en Java eran implementadas antes de la versión 1.5 pero sin el uso de clases genéricas. El uso de versiones anteriores de colecciones con colecciones genéricas es permitido por compatibilidad hacia atrás pero debe tenerse especial cuidado pues hay situaciones que el compilador no puede validar.

Iterator o la interfaz *ListIterator*. En general, para usar un iterador para recorrer una colección se debe: 1. Obtener un iterador al inicio de la colección llamando al método *iterator*() de la colección. 2. Definir un ciclo que haga la llamada a *hasNext*(). El ciclo iterará mientras el método sea verdadero. 3. En el ciclo, obtener cada elemento llamando al método *next*().

Ejemplo:

```
// Usando la interfaz Collection
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionTest {
        private static final String[] colors =
       { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
        private static final String[] removeColors =
       { "RED", "WHITE", "BLUE" };

    // crea ArrayList, añade Colors y la manipula
    public CollectionTest() {
            List< String > list = new ArrayList< String >();
        List< String > removeList = new ArrayList<String>();

        // añade elementos del arreglo colors a list
        for ( String color : colors )
            list.add( color );

       // añade elementos del arreglo removeColors a removeList
       for ( String color : removeColors )
            removeList.add( color );

       System.out.println( "ArrayList: " );
       // despliega contenido de list
       for ( int count = 0; count < list.size(); count++ )
            System.out.printf( "%s ", list.get( count ) );

       // remueve de list colores contenidos en removeList
       removeColors( list, removeList );

       System.out.println( "\n\nArrayList después de llamar removeColors: " );
       // despliega contenido de list
```

```
36          for ( String color : list )
37                System.out.printf( "%s ", color );
38       } // end CollectionTest constructor
39
40       // remueve colores especificados en collection2 de collection1
41       private void removeColors(
42          Collection< String > collection1, Collection< String > collection2 ) {
43                  // obtiene iterator
44                  Iterator< String > iterator = collection1.iterator();
45
46                  // mientras colección tiene elementos
47                  while ( iterator.hasNext() )
48                            if ( collection2.contains( iterator.next() ) )
49                                  iterator.remove(); // remueve color actual
50       }
51
52       public static void main( String args[] )  {
53             new CollectionTest();
54       }
55    }
```

Listing 36. Ejemplo usando la JCF en Java.

15.2.1 Ejemplos complementarios de Clases Genéricas en Java

Ejemplo de una pila genérica simple[3]:

```
1   import java.util.*;
2
3   public class PilaGenérica <T> {
4           private ArrayList<T> pila = new ArrayList<T> ();
5       private int tope = 0;
6
7       public int size () {
8                 return tope;
9       }
10
11      public void push (T elemento) {
12                pila.add (tope++, elemento);
13      }
```

[3]Basado en: Generic Stack

```
14
15      public T pop () {
16          return pila.remove (--tope);
17      }
18
19      public static void main (String[] args) {
20          PilaGenérica<Integer> p = new PilaGenérica<Integer> ();
21
22          p.push (17);
23          int i = p.pop ();
24          System.out.format ("%4d%n", i);
25       }
26   }
```

Listing 37. Ejemplo de una pila genérica simple en Java.

Ejemplo de una pila genérica como lista ligada[4]:

```
1   import java.util.Iterator;
2   import java.util.NoSuchElementException;
3
4   public class Pila<Elemento> implements Iterable<Elemento> {
5       private int tamaño;           // tamaño de la pila
6       private Nodo primer;      // tope de la pila
7
8       // clase anidada Nodo
9       private class Nodo {
10          private Elemento elemento;
11          private Nodo siguiente;
12      }
13
14      // Crea una pila vacia.
15      public Pila() {
16          primer = null;
17          tamaño = 0;
18      }
19
20      // Esta vacia la pila?
21      public boolean estaVacia() {
22          return primer == null;
23      }
```

[4]Ejemplo basado de: Stack

```
24
25      // Regresa el número de elementos en la pila
26      public int getTamaño() {
27          return tamaño;
28      }
29
30      // Añade elemento a la pila.
31      public void push(Elemento elemento) {
32          Nodo viejoPrimer = primer;
33
34       primer = new Nodo();
35          primer.elemento = elemento;
36          primer.siguiente = viejoPrimer;
37          tamaño++;
38      }
39
40     /**
41       *Regresa el elemento en el tope de la pila y lo elimina.
42       *Lanza una excepción si no hay elemento porque la pila este vacia.*/
43      public Elemento pop() {
44          if (estaVacia())
45                  throw new RuntimeException("Pila vacía");
46          Elemento elemento = primer.elemento;    // guarda elemento para retornarlo
47          primer = primer.siguiente;  // elimina el primer nodo
48          tamaño--;
49          return elemento;                  // regresa elemento
50      }
51
52     /*** Regresa el elemento en el tope de la pila sin modificarla.
53       * Lanza una excepción si la pila esta vacía.*/
54      public Elemento ver() {
55          if (estaVacia())
56                  throw new RuntimeException("Pila vacía");
57          return primer.elemento;
58      }
59
60     /*** Regresa representación en cadena.*/
61      public String toString() {
62          StringBuilder s = new StringBuilder();
63          for (Elemento elemento : this)
64              s.append(elemento + " ");
65          return s.toString();
```

```
66      }
67      //  Regresa un iterador a la pila que itera a través de los elementos en orden LIFO
68      public Iterator<Elemento> iterator()  {
69              return new ListIterator();
70              }
71    // Iterador, no se implementa remove() dado que es opcional
72      private class ListIterator implements Iterator<Elemento> {
73          private Nodo actual = primer;
74
75          public boolean hasNext()  {
76                  return actual != null;
77          }
78
79          public void remove() {
80                  throw new UnsupportedOperationException();
81          }
82
83          public Elemento next() {
84              if (!hasNext())
85                      throw new NoSuchElementException();
86              Elemento elemento = actual.elemento;
87              actual = actual.siguiente;
88              return elemento;
89          }
90      }
91
92      public static void main(String[] args) {
93          Pila<String> s = new Pila<String>();
94
95          String elemento1 = "un texto";
96          String elemento2 = "otro elemento";
97          String elemento3 = "xxxx";
98
99          s.push(elemento1);
100         s.push(elemento2);
101         s.push(elemento3);
102
103         while (!s.estaVacia()) {
104            System.out.println(s.pop());
105            System.out.println("(" + s.getTamaño() + " elemento(s) quedan en la pila)");
106         }
107     }
```

```
108 }
```

Listing 38. Ejemplo de una pila genérica como lista ligada en Java.

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 imprima todos los elementos en el almacenamiento.

Capítulo 16

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.

16.1 Manejo de Excepciones en Java

El modelo de excepciones de Java es similar al de C y C++, pero mientras en estos lenguajes no estamos obligados a manejar las excepciones, en Java es forzoso para el uso de ciertas clases; de lo contrario, el compilador generará un error.

16.1.1 ¿Cómo funciona?

Muchas tipos de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado hasta un intento de dividir por cero o intentar acceder a un arreglo fuera de sus límites. Cuando esto ocurre, la máquina virtual de Java crea un objeto de la clase *Exception* o *Error* y se notifica el hecho al sistema de ejecución. En este punto, se dice que se ha lanzado una excepción.

Un método se dice que es capaz de tratar una excepción si ha **previsto** el error que se ha producido y prevé también las operaciones a realizar para "recuperar" el programa de ese estado de error.

En el momento en que es lanzada una excepción, la máquina virtual de Java recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual de Java muestra un mensaje de error y el programa termina.

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción throw, lo que facilita la devolución de un código de error al método que invocó el método que causó el error.

Un ejemplo de una excepción generada (y no tratada) es el siguiente programa:

```
1 public class Excepcion {
2         public static void main(String argumentos[]) {
3                 int i=5, j=0;
4                 int k=i/j; // División por cero
5         }
6 }
```

Listing 39. Ejemplo de excepción en Java.

Al ejecutarlo, se verá que la máquina virtual Java ha detecta una condición de error y ha crea un objeto de la clase *java.lang.ArithmeticException*. Como el método donde se ha producido la excepción no es capaz de tratarla, es manejada

por la máquina virtual Java, que muestra un mensaje de error y finaliza la ejecución del programa.

16.1.2 Lanzamiento de excepciones (*throw*)

Como se ha comentado anteriormente, un método también es capaz de lanzar excepciones.

Sintaxis

```
método ( ) throws <lista de excepciones> {
        //código
        ...
        throw new <nombre Excepción>
        ...
}
```

donde $<$ *listadeexcepciones* $>$ es el nombre de cada una de las excepciones que el método puede lanzar.

Por ejemplo, en el siguiente programa se genera una condición de error si el dividendo es menor que el divisor:

Ejemplo:

```
1  public class LanzaExcepcion {
2          public static void main(String argumentos[]) throws ArithmeticException {
3
4                  int i=1, j=0;
5                  if (j==0)
6                          throw new ArithmeticException();
7                  else
8                          System.out.println(i/j);
9          }
10 }
```

Listing 40. Ejemplo lanzamiento de excepción en Java.

Para lanzar la excepción es necesario crear un objeto de tipo *Exception* o alguna de sus subclases (por ejemplo: *ArithmeticException*) y lanzarlo mediante la instrucción *throw*.

Los dos ejemplos vistos anteriormente, son capaces de lanzar una excepción en un momento dado, pero hasta aquí no difieren en mucho en su ejecución, ya que el resultado finalmente es la terminación del programa. En la siguiente sección se menciona como podemos darles un manejo especial a las excepciones, de tal forma que el resultado puede ser previsto por el programador.

16.1.3 Manejo de excepciones

En Java, de forma similar a C++ se pueden tratar las excepciones previstas por el programador utilizando unos mecanismos, los manejadores de excepciones, que se estructuran en tres bloques:

- El bloque *try*.
- El bloque *catch*.
- El bloque *finally*.

Concepto
Un **manejador de excepciones** es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar.

El bloque *try*

Lo primero que hay que hacer para que un método sea capaz de tratar una excepción generada por la máquina virtual Java o por el propio programa mediante una instrucción *throw*, es encerrar las instrucciones susceptibles de generarla en un bloque *try*.

```
1 try {
2 <instrucciones>
3 }
4 ...
```

Cualquier excepción que se produzca dentro del bloque *try* será analizada por el bloque o bloques *catch* que se verá en el punto siguiente. En el momento en que se produzca la excepción, se abandona el bloque *try* y, por lo tanto, las instrucciones que sigan al punto donde se produjo la excepción no serán ejecutadas.

El bloque catch

Cada bloque *try* debe tener asociado por lo menos un bloque *catch*.

Sintaxis

```
try {
        <instrucciones>
} catch (TipoExcepción1 nombreVariable1) {
<instruccionesBloqueCatch1>
} catch (TipoExcepción2 nombreVariable2) {
<instruccionesBloqueCatch2>
}
...
catch (TipoExcepciónN nombreVariableN) {
<instruccionesBloqueCatchN>
}
```

Por cada bloque *try* pueden declararse uno o varios bloques *catch*, cada uno de ellos capaz de tratar un tipo de excepción.

Para declarar el tipo de excepción que es capaz de tratar un bloque *catch*, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.

Ejemplo:

```
1  public class ExcepcionTratada {
2          public static void main(String argumentos[]) {
3                  int i=5, j=0;
4                  try {
5                          int k=i/j;
6                          System.out.println("Esto no se va a ejecutar.");
7                  }
8                  catch (ArithmeticException ex) {
9                          System.out.println("Ha intentado dividir por cero");
10                 }
11                 System.out.println("Fin del programa");
12         }
13 }
```

Listing 41. Ejemplo de excepción tratada en Java.

La ejecución se resuelve de la siguiente forma:

1. Cuando se intenta dividir por cero, la máquina virtual Java genera un objeto de la clase *ArithmeticException*.

2. Al producirse la excepción dentro de un bloque *try*, la ejecución del programa se pasa al primer bloque *catch*.

3. Si la clase de la excepción se corresponde con la clase o alguna subclase de la clase declarada en el bloque *catch*, se ejecuta el bloque de instrucciones catch y a continuación se pasa el control del programa a la primera instrucción a partir de los bloques *try-catch*.

También se podría haber utilizado en la declaración del bloque *catch*, una superclase de la clase *ArithmeticException*.

Por ejemplo:

```
catch (RuntimeException ex)
```

o

```
catch (Exception ex)
```

Sin embargo, es mejor utilizar excepciones más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error y si tratan de capturar todas las excepciones de una forma muy general, posiblemente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

El bloque *finally*

El bloque *finally* se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, **incluso** si no se produce ninguna excepción.

Este bloque **garantiza** que el código que contiene será ejecutado independientemente de que se genere o no una excepción:

Sintaxis

```
try {
        <instrucciones>
        }
catch (TipoExcepción1 nombreVariable1) {
        <instruccionesBloqueCatch1>
}

catch (TipoExcepción2 nombreVariable2) {
        <instruccionesBloqueCatch2>
}
        ...
catch (TipoExcepciónN nombreVariableN) {
        <instruccionesBloqueCatchN>
}
finally {
        <instruccionesBloqueFinally>
}
```

Es utilizado para no tener que repetir código en el bloque *try* y en los bloques *catch*. Este código sirve para llevar a buen término el bloque de código independientemente del resultado.

Veamos ahora la clase *ExcepcionTratada* con el bloque *finally*. Ejemplo:

```
1  public class ExcepcionTratada {
2          public static void main(String argumentos[]) {
3                  int i=5, j=0;
4                  try {
5                          int k=i /* /j */;//probar con y sin error
6                  }
7                  catch (ArithmeticException ex) {
8                          System.out.println("Ha intentado dividir por cero");
9                  }
10                 finally {
11                         System.out.println("Salida de finally");
12                 }
13                 System.out.println("Fin del programa");
```

```
14          }
15  }
```

Listing 42. Ejemplo de excepción con *finally* en Java.

Un ejemplo derivando la clase Exception de Java en un estilo similar al uso de la clase correspondiente en C++:

```
1   class DivisionByZeroException extends Exception {
2                   DivisionByZeroException(String msg) { super(msg); }
3   }
4
5   public class DivisionByZero {
6       public void division() throws DivisionByZeroException {
7           int num1 = 10;
8           int num2 = 0;
9
10          if (num2 == 0)
11                  throw new DivisionByZeroException("/ entre 0");
12          System.out.println(num1 + " / " + num2 + " = " + (num1 / num2));
13          System.out.println("terminando division().");
14      }
15
16      public static void main(String args[]) {
17           try {
18                   new DivisionByZero().division();
19           } catch (DivisionByZeroException e) {
20                   System.out.println("En main, tratando con " + e);
21           } finally {
22               System.out.println("Finally ejecutado en main.");
23           }
24           System.out.println("Finalizando main.");
25      }
26  }
```

Listing 43. Ejemplo derivando de *Exception* en Java.

16.1.4 Jerarquía de excepciones

Las excepciones son objetos pertenecientes a la clase *Throwable* o alguna de sus subclases.

Dependiendo del lugar donde se produzcan existen dos tipos de excepciones:

1. Las excepciones **síncronas** no son lanzadas en un punto arbitrario del programa sino que, en cierta forma, son previsibles en determinados puntos del programa como resultado de evaluar ciertas expresiones o la invocación de determinadas instrucciones o métodos.

2. Las excepciones **asíncronas** pueden producirse en cualquier parte del programa y no son tan previsibles. Pueden producirse excepciones asíncronas debido a dos razones:

   - La invocación del método *stop()* de la clase *Thread* que se está ejecutando.
   - Un error interno en la máquina virtual Java.

Dependiendo de si el compilador comprueba o no que se declare un manejador para tratar las excepciones, se pueden dividir en:

1. Las excepciones **comprobables** son repasadas por el compilador Java durante el proceso de compilación, de forma que si no existe un manejador que las trate, generará un mensaje de error.

2. Las excepciones **no comprobables** son la clase *RuntimeException* y sus subclases junto con la clase *Error* y sus subclases.

También pueden definirse por el programador subclases de las excepciones anteriores. Las más interesantes desde el punto de vista del programador son las subclases de la superclase *Exception* ya que éstas pueden ser **comprobadas** por el compilador.

La jerarquía completa de excepciones existentes en el paquete *java.lang* se puede consultar más adelante[1].

16.1.5 Excepciones definidas por el usuario

A partir de la jerarquía de excepciones definidas, es posible para el programador especificar su propia excepción tomando como base alguna de las excepciones de la jerarquía. Por ejemplo:

```
1 class MiExcepcion extends Exception{
2     String cad;
3
4     /* El constructor de nuestra excepción copia a una cadena
5     el mensaje que se pasa al lanzar la excepción
6     */
```

[1]Para un listado actual ver la documentación del jdk de Java más reciente.

```
7
8       MiExcepcion(String msj) {
9        cad=msj;
10      }
11      public String toString(){
12       return ("Se lanzó MiExcepcion: "+cad) ;
13      }
14  }
15
16  class EjemploMiExcepcion{
17      public static void main(String args[]){
18       try{
19           System.out.println("Iniciando bloque try");
20           // Lanzando mi propia excepción
21           throw new MiExcepcion("Mi mensaje de error");
22       }
23       catch(MiExcepcion exp){
24           System.out.println("Bloque catch") ;
25           System.out.println(exp) ;
26       }
27      }
28  }
```

Listing 44. Ejemplo de excepción definida por el programador.

16.1.6 Ventajas del tratamiento de excepciones

Las ventajas, mencionadas por Díaz-Alejo[2], de un mecanismo de tratamiento de excepciones como este son varias:

- Separación del código ”útil” del tratamiento de errores.
- Propagación de errores a través de la pila de métodos.
- Agrupación y diferenciación de errores mediante jerarquías.
- Claridad del código y obligación del tratamiento de errores.

[2]Díaz-Alejo Gómez, J.A., Programación con Java, IES Camp, Valencia, España, Last access: September 2006

16.1.7 Lista de Excepciones[3]

La jerarquía de clases derivadas de *Error* existentes en el paquete *java.lang* es la siguiente:

- java.lang.Object
  - java.lang.Throwable (implements java.io.Serializable)
- java.lang.Error
  - java.lang.AssertionError
  - java.lang.LinkageError
    - java.lang.ClassCircularityError
    - java.lang.ClassFormatError
      - java.lang.UnsupportedClassVersionError
    - java.lang.ExceptionInInitializerError
    - java.lang.IncompatibleClassChangeError
      - java.lang.AbstractMethodError
      - java.lang.IllegalAccessError
      - java.lang.InstantiationError
      - java.lang.NoSuchFieldError
      - java.lang.NoSuchMethodError
    - java.lang.NoClassDefFoundError
    - java.lang.UnsatisfiedLinkError
    - java.lang.VerifyError
  - java.lang.ThreadDeath
  - java.lang.VirtualMachineError
    - java.lang.InternalError
    - java.lang.OutOfMemoryError
    - java.lang.StackOverflowError
    - java.lang.UnknownError

La jerarquía de clases derivadas de *Exception* existentes en el paquete *java.lang* es la siguiente:

- java.lang.Object
  - java.lang.Throwable (implements java.io.Serializable)
  - java.lang.Exception
- java.lang.ClassNotFoundException
- java.lang.CloneNotSupportedException
- java.lang.IllegalAccessException
- java.lang.InstantiationException
- java.lang.InterruptedException
- java.lang.NoSuchFieldException

[3]Lista obtenida de la documentación del jdk en su versión 1.6

- java.lang.NoSuchMethodException
- java.lang.RuntimeException
  - java.lang.ArithmeticException
  - java.lang.ArrayStoreException
  - java.lang.ClassCastException
  - java.lang.EnumConstantNotPresentException
  - java.lang.IllegalArgumentException
    - java.lang.IllegalThreadStateException
    - java.lang.NumberFormatException
  - java.lang.IllegalMonitorStateException
  - java.lang.IllegalStateException
  - java.lang.IndexOutOfBoundsException
    - java.lang.ArrayIndexOutOfBoundsException
    - java.lang.StringIndexOutOfBoundsException
  - java.lang.NegativeArraySizeException
  - java.lang.NullPointerException
  - java.lang.SecurityException
  - java.lang.TypeNotPresentException
  - java.lang.UnsupportedOperationException

Las principales excepciones en otros paquetes Java son:

- class java.lang.Object
  - class java.lang.Throwable
    - class java.lang.Error
      - java.awt.AWTError
    - class java.lang.Exception
- java.io.IOException
- java.io.EOFException
- java.io.FileNotFoundException
- java.io.InterruptedIOException
- java.io.UTFDataFormatException
- java.net.MalformedURLException
- java.net.ProtocolException
- java.net.SocketException
- java.net.UnknownHostException
- java.net.UnknownServiceException
- RuntimeException
- java.util.EmptyStackException
- java.util.NoSuchElementException
- java.awt.AWTException

Parte II

Más allá de los Objetos

Capítulo 17

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

17.1 Afirmaciones en Java

Las afirmaciones fueron introducidas en Java desde la versión 1.4 del jdk. Cada afirmación debe contener una expresión booleana (*boolean* o *Boolean*).

Sintaxis

```
assert Expression1;
```

o

```
assert Expression1 : Expression2 ;
```

donde *Expression1* es una expresión booleana. Esta expresión es la evaluada y si es falsa la excepción *AssertionError* es lanzada. *Expression2* es una expresión que devuelve un valor (no *void*) que generalmente es usado para proveer de un mensaje para la excepción *AssertionError*.

17.1.1 Usando afirmaciones

Es importante no introducir código en las afirmaciones que en realidad sea una acción del programa. Por ejemplo:

```
assert ++i < max;
```

Es inapropiado pues se esta modificando el estado del programa al mismo tiempo que validando. Lo correcto sería algo del estilo:

```
1 i++;
2 assert i < max;
```

Errores detectados con afirmaciones deben ser errores que no deben pasar. Es por esto que se lanza un subtipo de *Error* en lugar de un subtipo de *Exception*. Si falla la validación de una afirmación se asume un error grave que nunca debe pasar.

17.1.2 Habilitando y deshabilitando las afirmaciones

Por omisión, las afirmaciones están deshabilitadas en tiempo de ejecución. Para cambiar de un estado a otro deben aplicarse parámetros especiales en la ejecución de la máquina virtual de Java:

```
-enableassertions | -ea
-disableassertions | -da
```

Estos modificadores pueden no llevar a su vez argumentos, por lo que active o desactiva las afirmaciones para todas las clases, o pueden indicarse nombres de paquetes ó clases específicas:

```
-enablesystemassertions | -esa
-disablesystemassertions | -dsa.
```

Ejemplo:

```
1  //Recuerda habilitar el uso de afirmaciones
2  public class AssertionEjemplo {
3     public static void main(String argv[])  {
4        //obtener un número del primer argumento
5        int num = Integer.parseInt(argv[0]);
6
7            assert num <=10;  //se detiene si num>10
8
9        System.out.println("Pasó");
10    }
11 }
```

Listing 45. Ejemplo de afirmaciones en Java.

Capítulo 18

Pruebas de unidad

Capítulo 19

Multihilos: Introducción a la Programación Concurrente en Java

19.1 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].

19.2 Concurrencia

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

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

Creo que estaremos de acuerdo en que las **operaciones secuenciales** son aquellas que ocurren una después de otra; es decir, están ordenadas en el tiempo. De aquí se desprende que las **operaciones paralelas** son aquellas que ocurren al mismo tiempo. Es común hablar de paralelismo cuando hablamos de operaciones de hardware.

Sin embargo al hablar de concurrencia nos referimos por lo general al código fuente, donde un conjunto de operaciones son concurrentes si pueden ejecutarse en paralelo, lo que no quiere decir que obligatoriamente se ejecuten así. Es por eso que podemos tener procesos concurrentes sin tener hardware paralelo. Por ejemplo, las computadoras con *Windows* que realizan varios procesos al mismo tiempo como mandar a imprimir, bajar un archivo de Internet, enviar un *eMail*, etc. Las PC's son por lo general de un solo procesador y no cuentan con procesamiento paralelo; sin embargo, estamos realizando procesos concurrentes. Lo mismo sucede con todas las máquinas que tienen un solo procesador y cuentan con sistema operativo multitareas como *Unix* o *Linux*[2]. Podemos decir que la concurrencia es cuando un conjunto de instrucciones no depende de otro conjunto y no importa el orden de ejecución entre ellas: son **potencialmente paralelas**.

19.3 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 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.

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

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

[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.

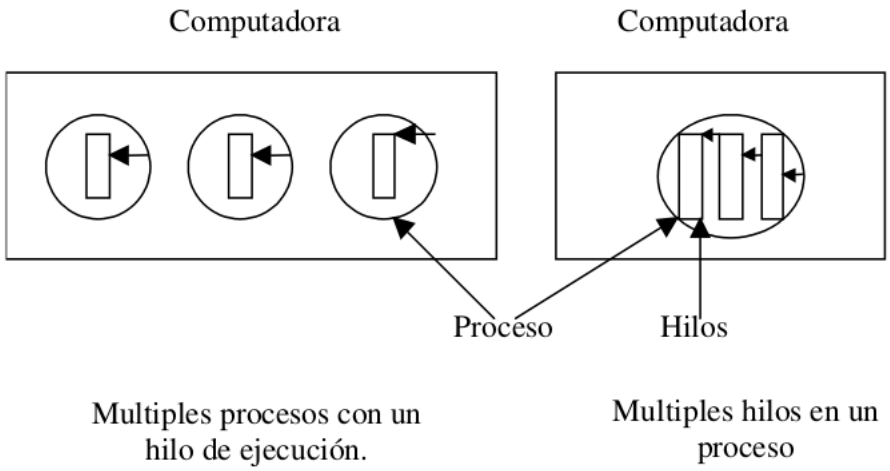


Fig. 19.1. Múltiples procesos vs. múltiples hilos en un proceso

19.4 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*.

19.4.1 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.

19.4.2 Programas de flujo múltiple

Los programas en Java implementan un flujo único de manera implícita. Sin embargo, Java posibilita la creación y control de hilos explícitamente. La utilización de hilos en Java, permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar hilos, permite que se puedan implementar muy poderosas y portables aplicaciones.

Las aplicaciones multihilos utilizan muchos contextos de ejecución para cumplir su trabajo. Hacen uso del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo para cada 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 }
```

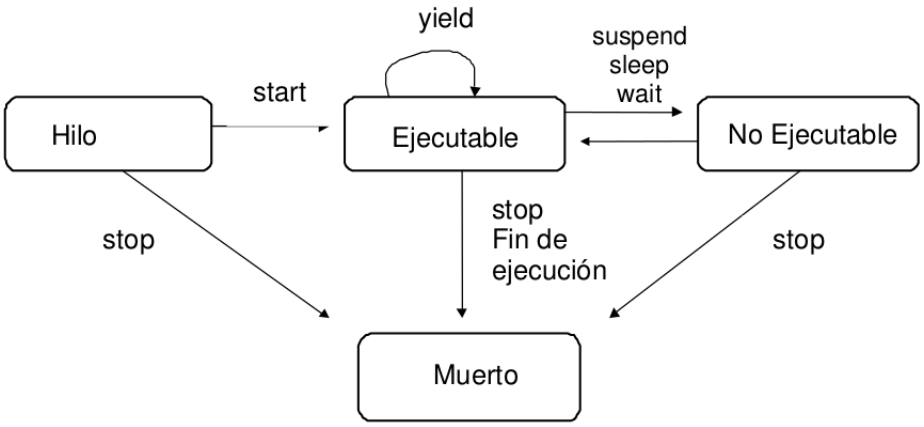


Fig. 19.2. Estados de un hilo en Java

Listing 46. 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.

19.5 Estados de un hilo

Cada hilo de ejecución en Java, es un objeto que puede estar en diferentes estados.

En la figura 19.2 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

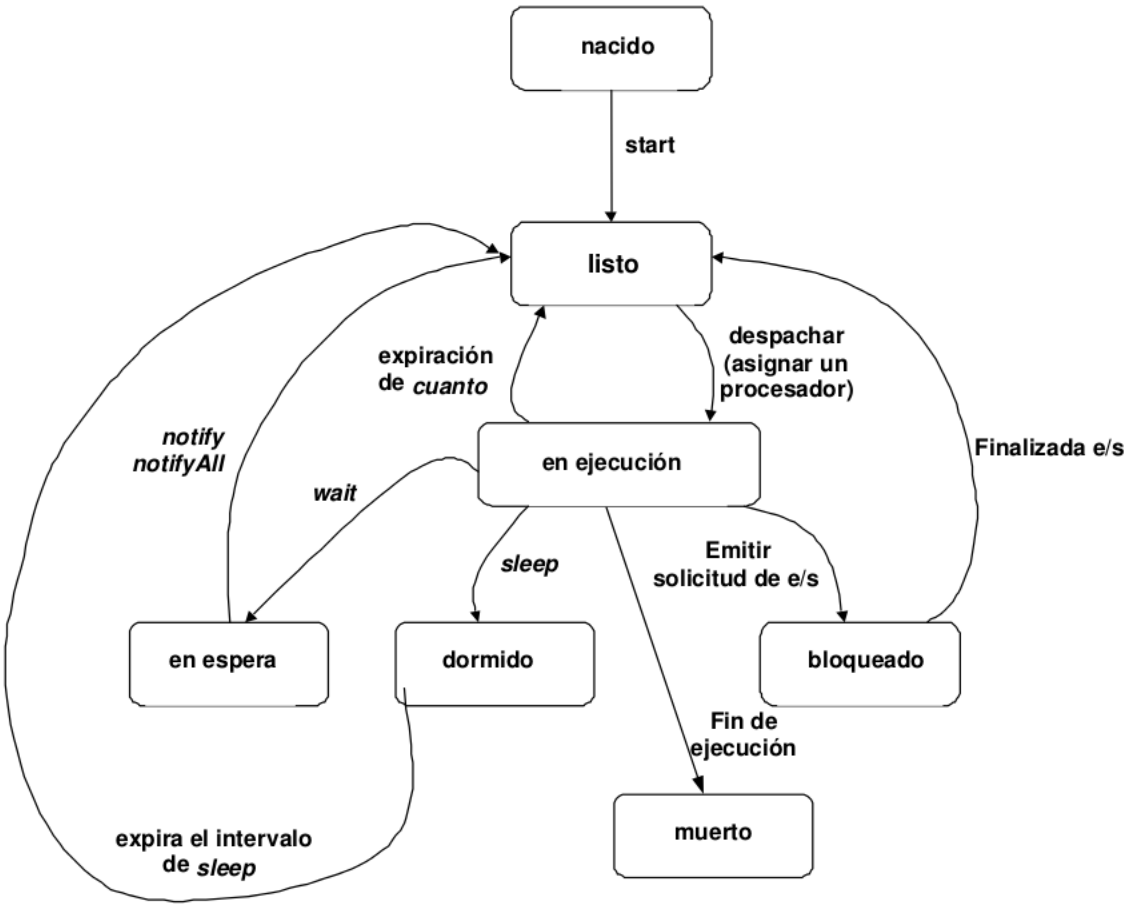


Fig. 19.3. Estados de un hilo en Java (2)

para la ejecución de los programas concurrentes[5].

Veamos ahora un diagrama (figura 19.3) que muestra de manera más completa los estados en los que puede estar un hilo.

19.6 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.

```
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*

19.7 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) {}
17          }
18      }
19   }
20
21 public class prioridadHilo  {
22
23
24    public static void main(String args[])
25      {
26        miHilo hilo_min = new miHilo ("Hilo Min");
27        miHilo hilo_max = new miHilo ("Hilo Max");
28
29        hilo_min.setPriority(Thread.MIN_PRIORITY);
30        hilo_max.setPriority(Thread.MAX_PRIORITY);
31        hilo_min.start();
32        hilo_max.start();
33      }
34   }
```

Listing 47. 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:

```
1  // Muestra múltiples hilos desplegándose a diferentes
2  // intervalos.
3
4  public class PrintTest {
5     public static void main( String args[] )
6     {
7        PrintThread thread1, thread2, thread3, thread4;
8
9        thread1 = new PrintThread( "1" );
10       thread2 = new PrintThread( "2" );
11       thread3 = new PrintThread( "3" );
12       thread4 = new PrintThread( "4" );
13
14       thread1.start();
```

```
15          thread2.start();
16          thread3.start();
17          thread4.start();
18       }
19    }
20
21    class PrintThread extends Thread {
22       int sleepTime;
23
24       // constructor PrintThread asigna nombre al hilo
25       // llamando al constructor de Thread
26       public PrintThread( String id )
27       {
28          super( id );
29
30          // valor aleatorio para dormir el hilo de 0 a 5 segundos
31          sleepTime = (int) ( Math.random() * 5000 );
32
33          System.out.println( "Nombre: " + getName() +
34                              ";  durmiendo: " + sleepTime );
35       }
36
37       // ejecuta el hilo
38       public void run()
39       {
40          try {
41             sleep( sleepTime );
42          }
43          catch ( InterruptedException exception ) {
44             System.err.println( "Excepcion: " +
45                                 exception.toString() );
46          }
47          System.out.println( "Hilo " + getName() );
48       }
49    }
```

Listing 48. Ejemplo múltiples hilos desplegándose a diferentes // intervalos.

19.8 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
```

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

```
20 public class MultiHilo2 {
21     public static void main(String arg[]) {
22         HiloEterno infinito = new HiloEterno("Al infinito");
23         HiloEterno  masAlla = new HiloEterno("y mas alla");
24         infinito.start();
25         masAlla.start();
26     }
27 }
28
```

Listing 49. Ejemplo que implementa un hilo y cede el control a otros hilos.

Capítulo 20

Multihilos: Programación Concurrente en Java (2)

20.1 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] 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

```
1  class Cuenta {
2
3    static int balance = 1000;
4    static int gasto = 0;
5
6     static public synchronized void retirar(int cantidad)
7       {
8         if (cantidad <= balance)
9           {
10            System.out.println("cheque: " + cantidad);
11            balance -= cantidad;
12            gasto += cantidad;
13            System.out.print("balance: " + balance);
14            System.out.println(", se gastó: " + gasto);
15          }
16        else
17          {
18            System.out.println("rebotó: " + cantidad);
19          }
20      }
21   }
22
23 class miHilo extends Thread {
24    public void run()
25      {
26        for (int i = 0; i < 10; i++)
27          {
28            try {
29               sleep(100);
30              }
31            catch (InterruptedException e) {}
32            Cuenta.retirar((int) (Math.random() * 500 ));
33          }
34      }
35   }
36
37 public class Sincronizar  {
38
39         public static void main(String arg[]){
40       new miHilo().start();
41       new miHilo().start();
```

```
42      }
43  }
44
```

Listing 50. 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 51. Ejemplo de multihilos declarando un bloque de código sincronizado.

20.1.1 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 );
```

[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.

```
6        ConsumeInteger c = new ConsumeInteger( h );
7
8        p.start();
9        c.start();
10    }
11 }
12
13 class ProduceInteger extends Thread {
14    private HoldInteger pHold;
15
16    public ProduceInteger( HoldInteger h )
17    {
18       pHold = h;
19    }
20
21    public void run()
22    {
23       for ( int count = 0; count < 10; count++ ) {
24          pHold.setSharedInt( count );
25          System.out.println( "El productor puso: " +
26                              count );
27
28          try {
29             sleep( (int) ( Math.random() * 3000 ) );
30          }
31          catch( InterruptedException e ) {
32             System.err.println( "Excepcion " + e.toString());
33          }
34       }
35    }
36 }
37
38 class ConsumeInteger extends Thread {
39    private HoldInteger cHold;
40
41    public ConsumeInteger( HoldInteger h )
42    {
43       cHold = h;
44    }
45
46    public void run()
47    {
```

```
48        int val;
49
50        val = cHold.getSharedInt();
51        System.out.println( "Recuperado por el consumidor: " + val );
52
53        while ( val != 9 ) {
54
55           try {
56              sleep( (int) ( Math.random() * 3000 ) );
57           }
58           catch( InterruptedException e ) {
59              System.err.println( "Excepcion " + e.toString() );
60           }
61
62           val = cHold.getSharedInt();
63           System.out.println( "Recuperado por el consumidor: " + val );
64        }
65     }
66  }
67
68  class HoldInteger {
69     private int sharedInt;
70
71     public void setSharedInt( int val ) {
72      sharedInt = val;
73     }
74
75     public int getSharedInt() {
76      return sharedInt;
77     }
78  }
```

Listing 52. 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 53. 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 54. Ejemplo adicional de productor/consumidor.

Esperando que haya quedado claro el uso de *wait()* y *notify()*, mencionaremos algunas observaciones que consideramos importantes. En primer lugar, un hilo que es puesto en espera mediante el método *wait()*, debe ser notificado en algún momento de que puede continuar, de lo contrario puede quedarse esperando indefinidamente. Es vital entonces que para cada llamada a *wait()* haya una correspondiente llamada a *notify()*. Un hilo también puede ser mandado a la cola de espera porque trate de entrar un método sincronizado y ya se encuentre en ejecución un método sincronizado para ese objeto; pero, a diferencia de un hilo puesto en espera explícitamente, este hilo se reactivará cuando tenga oportunidad de entrar sin necesidad de que le notifiquen.

El método *notify()* sólo le indica a un hilo que puede dejar de esperar. Existe además un método *notifyAll()*, que hace que todos los hilos que están en espera

traten de ejecutarse. Obviamente sólo uno podrá hacerlo, pero todos tendrán la oportunidad de entrar.

Hay que tener cuidado con las sincronizaciones, ya que demasiado código sincronizado puede hacer muy lenta la ejecución de los hilos. Se debe sincronizar únicamente cuando se trate de situaciones críticas que tienen que ver con datos compartidos.

Un problema común es que un programa muy sincronizado genere una ejecución casi secuencial. Otro problema más grave es que puede llegarse a dar el caso de que todos los hilos se queden esperando, esto se conoce como **abrazo mortal** (*deadlock*), y por supuesto que hay que tratar de evitarlo[4]. El abrazo mortal se puede evitar con una cuidadosa programación que promueva el uso ordenado de los recursos[5].

20.2 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");
```

20.3 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

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

nosotros sea considerado *daemon* debe especificarse mediante el uso del método *setDaemon()*:

```
1 Thread miHilo= new Thread();
2 miHilo.setDaemon(true);
```

Es importante señalar que un hilo debe de ser determinado como *daemon* antes de que se ejecute el método *start()* de ese hilo, o será lanzada la excepción *IllegalThreadStateException*.

20.4 La interfaz *Runnable*

Se ha visto la implementación de hilos heredando de la clase *Thread*, la cual es la forma más común. Pero puede ser que la clase, a la que queremos implementar funcionalidades de hilos, ya tenga definida herencia de otra clase. Como Java no soporta herencia múltiple, para implementar el hilo en la clase tendría que hacerse a través de la interfaz *Runnable*.

Esta interfaz únicamente tiene definido el método *run()*, el cual debe implementarse al igual que en la clase que se extiende la clase Thread. El esquema general para usar la interfaz *Runnable* es:

```
1 public class MiHilo implements Runnable {
2
3   public void run(){
4      //código del hilo
5   }
6 }
```

El inicio de la ejecución de un hilo difiere cuando se implementa esta interfaz, ya que se debe crear una instancia de la clase *Thread* y pasarle como parámetro al constructor del hilo un objeto de la clase que implementa la interfaz *Runnable*:

```
1 Thread miHilo= new Thread( new MiHilo() );
2
3 miHilo.start();
```

Ejemplo:

```
1  public class DemoThread implements Runnable {
2
3      @Override
4      public void run() {
5          for (int i = 0; i < 3; i++) {
6              System.out.println("Hilo Hijo ");
7
8              try {
9                  Thread.sleep(200);
10             } catch (InterruptedException ie) {
11                 System.out.println("Hilo hijo se ha interrumpido! " + ie);
12             }
13         }
14
15         System.out.println("Hilo hijo finalizado!");
16     }
17
18     public static void main(String[] args) {
19         Thread t = new Thread(new DemoThread ());
20
21         t.start();
22
23         for (int i = 0; i < 3; i++) {
24             System.out.println("Hilo principal");
25             try {
26                 Thread.sleep(200);
27             } catch (InterruptedException ie) {
28                 System.out.println("Hilo hijo interrumpido! " + ie);
29             }
30         }
31         System.out.println("Hilo principal finalizado!");
32     }
33 }
```

Listing 55. Ejemplo de multihilos usando la interfaz Runnable.

Ejemplo:[7]

```
1  // Ejemplo de implementacion de la interfaz Runnable
2  import java.awt.*;
```

[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().

```
3  import java.applet.Applet;
4
5  public class RandomCharacters extends Applet
6                                 implements Runnable {
7
8     String alphabet;
9     TextField output1, output2, output3;
10    Button button1, button2, button3;
11
12    Thread thread1, thread2, thread3;
13
14    boolean suspend1, suspend2, suspend3;
15
16    public void init()
17    {
18       alphabet = new String( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
19       output1 = new TextField( 10 );
20       output1.setEditable( false );
21       output2 = new TextField( 10 );
22       output2.setEditable( false );
23       output3 = new TextField( 10 );
24       output3.setEditable( false );
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
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 56. Ejemplo de implementacion de la interfaz *Runnable* con un *applet*.

20.5 Actividades sugeridas

Hacer un programa que simule el problema de los filósofos tragones (ver figura 20.1).

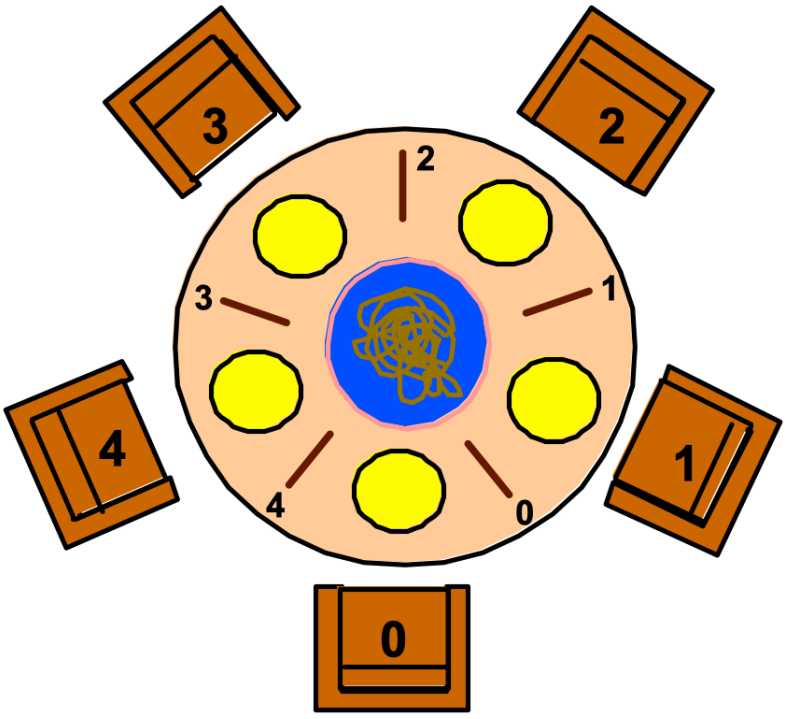


Fig. 20.1. Filósofos tragones

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 21

Java: Clases anidadas y anónimas

21.1 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

[1]Fuente: Nested class

no tienen acceso a los otros miembros de la clase externa. Las clases anidadas, al ser miembros de una clase pueden ser definidas como públicas, protegidas o privadas (privadas a nivel de paquete).

Una clase anidada puede ser

- Es una forma de agrupar clases que son usadas en un sólo lugar
- Incrementa la encapsulación
- Puede llevar a un código más fácil de leer y/o mantener.

Ejemplo:Clases internas.

```
1  public class EstructuraDeDatos {
2
3      // Crea un arreglo
4      private final static int TAMAÑO = 15;
5      private int[] arregloDeEnteros = new int[TAMAÑO];
6
7      public EstructuraDeDatos() {
8          for (int i = 0; i < TAMAÑO; i++) {
9                  arregloDeEnteros[i] = i;
10         }
11     }
12
13     public void despliegaPares() {
14
15         // despliega valores de índices pares del arreglo
16             IteradorDeEstructuraDeDatos iterador = this.new IteradorPares();
17         while (iterador.hasNext()) {
18             System.out.print(iterador.next() + " ");
19         }
20         System.out.println();
21     }
22
23     //interfaz anidada extiende la interfaz Iterator<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 57. 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 58. Ejemplo variables con el mismo nombre en clases internas .

21.2 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");
```

[2] Anonymouse classes

```
14              }
15              public void saludarAlguien(String alguien) {
16                  nombre = alguien;
17                  System.out.println("Hello " + nombre);
18              }
19          }
20
21          HolaMundo saludarEnInglés = new SaludarEnInglés();
22
23          HolaMundo saludarEnFrances = new HolaMundo() {
24              String nombre = "tout le monde";
25              public void saludar() {
26                      saludarAlguien("tout le monde");
27              }
28              public void saludarAlguien(String alguien) {
29                  nombre = alguien;
30                  System.out.println("Salut " + nombre);
31              }
32          };
33
34          HolaMundo saludarEnEspañol = new HolaMundo() {
35              String nombre = "mundo";
36              public void saludar() {
37                      saludarAlguien("mundo");
38              }
39              public void saludarAlguien(String alguien) {
40                  nombre = alguien;
41                  System.out.println("Hola, " + nombre);
42              }
43          };
44
45          saludarEnInglés.saludar();
46          saludarEnFrances.saludarAlguien("Juliette");
47          saludarEnEspañol.saludar();
48      }
49
50      public static void main(String... args) {
51                      ClasesAnónimasHolaMundo miApl =
52              new ClasesAnónimasHolaMundo();
53              miApl.decirHola();
54      }
```

```
55 }
```

Listing 59. Ejemplo de clases anónimas.

Capítulo 22

Java: Expresiones lambda λ

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

22.2 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.

[2] Lambda expressions java tutorial

- El tipo del parámetro puede indicarse explícitamente o puede ser inferido del contexto.
- Cuando no hay parámetros se debe indicar con paréntesis vacíos (). Si se tienen un sólo parámetro y su tipo es inferido, los paréntesis son opcionales.
- El cuerpo de la expresión lambda puede ir vacío.
- Si se tiene una sola instrucción en el cuerpo, se pueden omitir las llaves y el tipo de retorno de la función es el mismo que el de la instrucción.

Las expresiones lambda pueden ser usadas en lugar de las clases anónimas para implementar el método de una interfaz funcional. Una **interfaz funcional** es una interfaz que tiene sólo un método abstracto declarado. Cada expresión lambda puede ser implícitamente asignada a una interfaz funcional.

Por ejemplo, inclusive cuando no especificamos la interfaz funcional, el compilador automáticamente resuelve que la siguiente expresión lambda puede ser enmascarada a la interfaz *Runnable* en la firma del constructor *Thread(Runnable r)*:

```
1 new Thread(
2         ( ) -> System.out.println("ejemplo de expresión lambda")
3 ).start();
```

Dos diferencias importantes entre expresiones lambda y clases anónimas:

- Para una clase anónima el uso de this se refiere a la clase anónima, mientras que el uso de this en una expresión lambda se refiere a la clase que contiene la expresión lambda.
- Para el compilador, una clase anónima es tratada como una clase y generará su código respectivo, mientras que una expresión lambda es compilada y convertida en un método privado de la clase que contiene la expresión.

Capítulo 23

Java: Interfaz gráfica con JavaFX

23.1 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 *multi-touch* (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 23.1 [3].

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

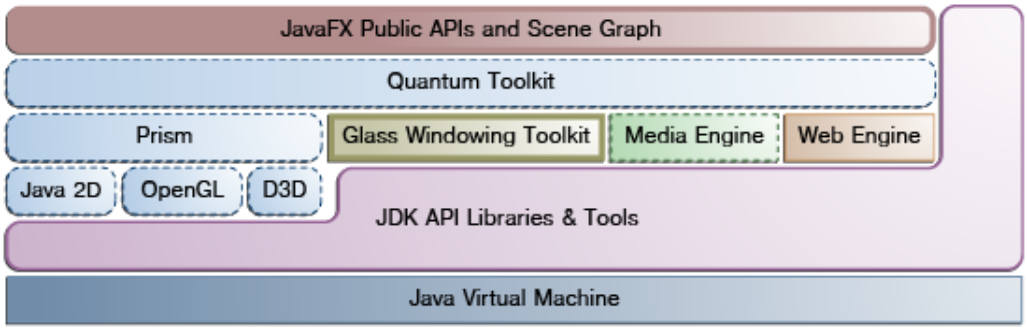


Fig. 23.1. Arquitectura de JavaFX

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:

- *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 60. 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 61. 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 escenarios. Puede verse a un escenario como una plataforma que soporta escenas y nodos como actores que intervienen en las escenas.

Un ejemplo[**?**] con múltiples escenarios, se omite el método *main* ya que, como se explicó no debe ser necesario:

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MúltipleEscenario extends Application {
7      @Override
8      public void start(Stage escenarioPrimario) {
9          // Crea una escena y coloca un botón en la escena
10         Scene escena = new Scene(new Button("OK"), 200, 250);
11         escenarioPrimario.setTitle("MúltipleEscenarioApp"); // asigna título al escenar
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 62. Ejemplo de JavaFX con múltiples escenarios.

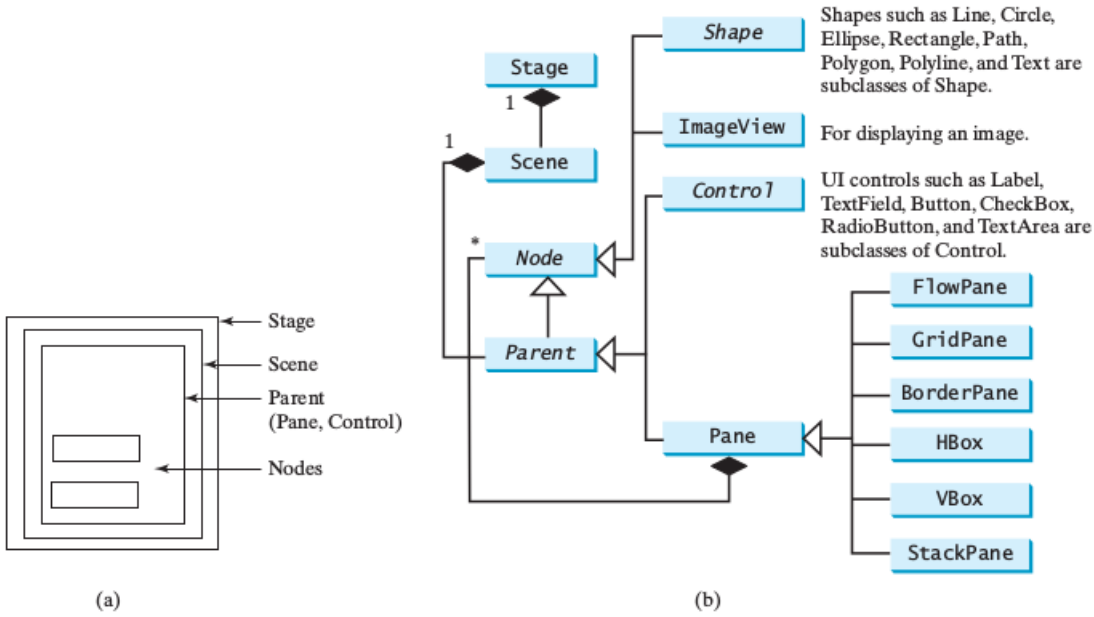


Fig. 23.2. (a) Paneles son usados para mantener nodos. (b) Nodos pueden ser figuras, vstas de imágenes, controles IU, y paneles.

23.2 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 23.2 [**?**].

Un ejemplo [**?**] con panel:

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5  import javafx.scene.layout.StackPane;
6
7  public class BotónEnPanel extends Application {
8      @Override
9      public void start(Stage primaryStage) {
10         // Crea una escena  y coloca botón en la escena
```

```
11          StackPane panel = new StackPane();
12          panel.getChildren().add(new Button("OK"));
13          Scene escena = new Scene(panel, 200, 50);
14          primaryStage.setTitle("Botón en un panel");
15          primaryStage.setScene(escena);
16          primaryStage.show();
17      }
18  }
```

Listing 63. 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.

23.2.1 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:

```java
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class MuestraFlowPane extends Application {
    @Override
    public void start(Stage escenarioPrimario) {
        // Crea un panel y asigna sus propiedades
        FlowPane panel = new FlowPane();
                // asigna las propiedades de relleno de espacio
                // con un objeto de Insets
                //Construye una nueva instancia Insets
                //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.h
        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 64. Ejemplo JavaFX de panel usando *FlowPane*.

Usando BorderPane. Ejemplo:

```
1  import javafx.application.Application;
2  import javafx.geometry.Insets;
3  import javafx.scene.Scene;
4  import javafx.scene.control.Label;
5  import javafx.scene.layout.BorderPane;
6  import javafx.scene.layout.StackPane;
7  import javafx.stage.Stage;
8
9  public class MuestraBorderPane extends Application {
10     @Override
11     public void start(Stage escenarioPrimario) {
12         // Crea un BorderPane
13         BorderPane panel = new BorderPane();
14         // Coloca nodos en el panel
15         panel.setTop(new PanelAdaptado("Arriba"));
16         panel.setRight(new PanelAdaptado("Derecha"));
17         panel.setBottom(new PanelAdaptado("Abajo"));
18         panel.setLeft(new PanelAdaptado("Izquierda"));
19         panel.setCenter(new PanelAdaptado("Centro"));
20
21         // Crea una escena y la coloca en el escenario
22         Scene escena = new Scene(panel);
23         escenarioPrimario.setTitle("MuestraBorderPane");
24         escenarioPrimario.setScene(escena);
25         escenarioPrimario.show();
26     }
27 }
28
29 // Define un panel adaptado para mantener una etiqueta en el centro del panel
30 class PanelAdaptado extends StackPane {
31     public PanelAdaptado(String title) {
32         getChildren().add(new Label(title));
33         setStyle("-fx-border-color: red");
34         setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
35     }
36 }
```

Listing 65. 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 24

Java: Manejo de eventos con JavaFX

24.1 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 24.1

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.

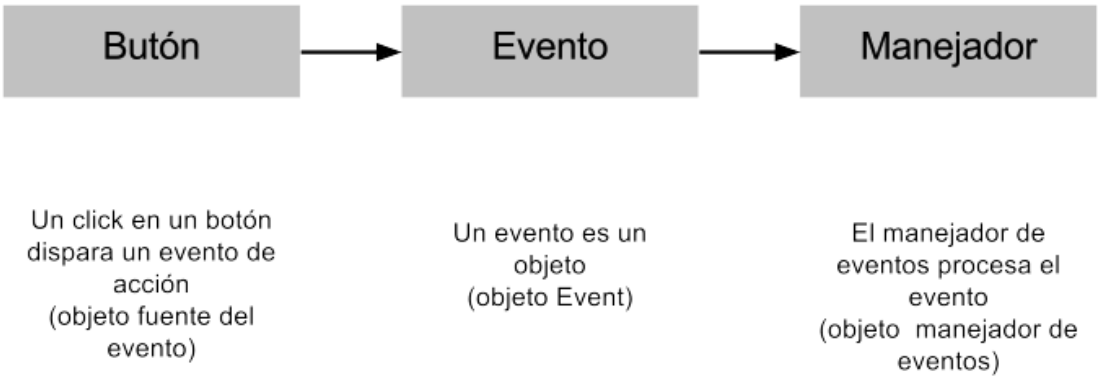


Fig. 24.1. Proceso de manejo de evento

Por su parte, *<T extends Event>* especifica que *T* es un tipo genérico que es un subtipo de *Event*.

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

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 66. 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.

24.2 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.

24.3 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 67. 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 68. Ejemplo JavaFX dos botones con manejo de eventos.

24.3.1 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();
```

[1] Anonymous classes

```
15          btn.setText("Decir 'Hola'");
16
17          //clase anónima dentro de setOnAction
18          btn.setOnAction(new EventHandler<ActionEvent>() {
19
20              @Override
21              public void handle(ActionEvent event) {
22                  System.out.println("¡Hola!");
23              }
24          });
25
26          StackPane raíz = new StackPane();
27          raíz.getChildren().add(btn);
28          escenarioPrimario.setScene(new Scene(raíz, 300, 250));
29          escenarioPrimario.show();
30      }
31  }
```

Listing 69. 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 70. Ejemplo JavaFX con clases anónimas y uso de *TextField*.

24.3.2 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 71. Ejemplo JavaFX y manejador de eventos con expresiones lambda.

24.3.3 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();
```

```
        Text texto = new Text(20, 20, "Presiona y arrastra un texto cualquiera");
        panel.getChildren().addAll(texto);

        texto.setOnMouseDragged(e -> {
            texto.setX(e.getX());
            texto.setY(e.getY());
        });

        Scene escena = new Scene(panel, 300, 100);
        escenarioPrimario.setTitle("Ejemplo de Evento de Mouse");
        escenarioPrimario.setScene(escena);
        escenarioPrimario.show();
    }
}
```

Listing 72. Ejemplo JavaFX con evento de mouse.

24.3.4 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 73. Ejemplo JavaFX con evento de teclado.

24.3.5 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);
```

```
26          hBox.getChildren().add(btReducir);
27
28          // Crear y registrar manejadores
29          btAgrandar.setOnAction(e -> panelCírculo.agrandar());
30          btReducir.setOnAction(e -> panelCírculo.reducir());
31
32          panelCírculo.setOnMouseClicked(e -> {
33              if (e.getButton() == MouseButton.PRIMARY) {
34                  panelCírculo.agrandar();
35              } else if (e.getButton() == MouseButton.SECONDARY) {
36                  panelCírculo.reducir();
37              }
38          });
39
40          panelCírculo.setOnKeyPressed(e -> {
41              if (e.getCode() == KeyCode.U) {
42                  panelCírculo.agrandar();
43              } else if (e.getCode() == KeyCode.D) {
44                  panelCírculo.reducir();
45              }
46          });
47
48          BorderPane panelBorde = new BorderPane();
49          panelBorde.setCenter(panelCírculo);
50          panelBorde.setBottom(hBox);
51          BorderPane.setAlignment(hBox, Pos.CENTER);
52
53          Scene escena = new Scene(panelBorde, 200, 150);
54          escenarioPrimario.setTitle("EjemploEventoTecladoMouse");
55          escenarioPrimario.setScene(escena);
56          escenarioPrimario.show();
57          panelCírculo.requestFocus();
58      }
59  }
60
61  // vista en ejemplo anterior
62  class PanelCírculo extends StackPane {
63      private Circle círculo = new Circle(50);
64
65      public PanelCírculo() {
66          getChildren().add(círculo);
67          círculo.setStroke(Color.BLACK);
```

```
68          círculo.setFill(Color.WHITE);
69      }
70
71      public void agrandar() {
72          círculo.setRadius(círculo.getRadius() + 2);
73      }
74
75      public void reducir() {
76          círculo.setRadius(círculo.getRadius() > 2 ?
77              círculo.getRadius() - 2 : círculo.getRadius());
78      }
79  }
```

Listing 74. Ejemplo JavaFX con evento de teclado y mouse.

24.3.6 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 75. Ejemplo JavaFX con un objeto observable.

Capítulo 25

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.

25.1 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 25.1 muestra una lista de protocolos de uso común en Internet.

Cuadro 25.1. 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)

El paquete *java.net* cuenta hasta la versión 1.4 del jdk con 6 interfaces, 27 clases y 11 excepciones. Las clases más usadas son:

1. *URL*. Representa un URL de Internet.
2. *URLConnection*. Es un complemento de URL (no una subclase de ella). Cubre algunas operaciones más complejas.
3. *Socket*. Establece conexiones TCP/IP.
4. *DatagramPacket*. Establece conexiones de tipo UDP.
5. *InetAddress*. Representa una dirección IP.

25.1.1 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       }
```

[1]URL. *Uniform Resource Locator*

```
21
22    public boolean mouseDown(Event evt, int x, int y)
23      {
24        getAppletContext().showDocument(utm);
25        return(true);
26      }
27    }
```

Listing 76. 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.

25.1.2 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 77. Ejemplo de *InetAddress*, obtiene la dirección IP de la máquina local.

[2]No se pruebe en el *appletviewer* ya que este no despliega más que el *applet* y no muestra la página html.

El programa anterior usa una instancia de *InetAddress* para obtener la dirección de la máquina local mediante el método *getLocalHost*() de la clase. Es un ejemplo muy simple del uso de la clase.

El siguiente programa recibe como parámetro el nombre de un servidor y obtiene la dirección asociada a ese nombre.

Ejemplo:

```
1  //identifica la direccion IP asociada al host
2  import java.net.InetAddress;
3  import java.net.UnknownHostException;
4  import java.lang.System;
5
6  public class NSLookupApp {
7   public static void main(String args[]) {
8    try {
9     if(args.length!=1){
10     System.out.println("Sintaxis: java NSLookupApp nombreServidor");
11     return;
12    }
13    InetAddress host = InetAddress.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 78. Ejemplo de *InetAddress,* identifica la dirección IP asociada al *host*.

Ejecutando por ejemplo:

```
\$java NSLookupApp www.utm.mx
```

25.1.3 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.

25.1.4 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 79. Ejemplo de programa cliente servidor con sockets .

El programa muestra a un hilo cliente solicitando una cadena de un flujo de datos al servidor: cada vez que se da *click* sobre la ventana el servidor inicia a un hilo cliente que a su vez se comunica con el servidor.

En el **sección complementaria uno** se muestra otro ejemplo de uso de la clase *Socket* para crear un cliente y en la **sección complementaria dos** se muestra el código correspondiente al servidor. Estos programas se comunican entre sí: el cliente es capaz de enviar una cadena y recibirla de vuelta modificada por el servidor. Cada uno corre de manera independiente e idealmente debería ser probado en distintas máquinas en red.

Ejemplo de ejecución del **cliente**:

```
1 $ java PortTalkApp yodocono.utm.mx 1234
2 Conectando a: yodocono.utm.mx puerto 1234.
3 servidor destino yodocono.utm.mx.
4 IP servidor destino 192.100.170.5.
5 numero de puerto servidor destino 1234.
6 servidor Local iec23.
7 IP servidor Local 192.100.170.33.
8 numero de puerto servidor Local 2162.
9 Enviar, recibir, o salir (E/R/S): e
```

```
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
```

25.1.5 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*.

25.1.6 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*:

[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.

```
$ 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.
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.

25.2 Complemento 1. *PortTalkApp*

```
//Ejemplo de uso de la clase Socket
import java.lang.System;
import java.net.Socket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.io.*;

public class PortTalkApp {
 public static void main(String args[]){
  PortTalk portTalk = new PortTalk(args);
  portTalk.displayDestinationParameters();
  portTalk.displayLocalParameters();
  portTalk.chat();
  portTalk.shutdown();
 }
}

class PortTalk {
 Socket connection;
 DataOutputStream outStream;
 BufferedReader inStream;
 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 80. Ejemplo de uso de la clase Socket. *PortTalkApp*.

25.3 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 81. Ejemplo de uso de la clase *ServerSocket*.

25.4 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 82. Ejemplo de escucha de un socket UDP. *TimeServerApp*

25.5 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 83. 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 26

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.

26.1 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.

26.2 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.

26.2.1 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 84. 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:

[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.

```java
//Ejemplo de  cliente de SMTP
import java.io.*;
import java.net.*;

public class CorreoJava {
   static PrintStream ps = null;        // envío de mensajes
   static BufferedReader dis =null;


   public static void enviar(String str) throws IOException
     {
       ps.println(str); // enviar una cadena SMTP
       ps.flush();      // vaciar la cadena

       System.out.println("Java envió: " + str);
     }

   public static void recibir() throws IOException
     {
       String readstr = dis.readLine(); // obtener la respuesta SMTP
       System.out.println("respuesta SMTP: " + readstr);
     }

   public static void main (String args[])
     {
       String HELO = "HELO ";
       String MAIL_FROM = "MAIL FROM: miCuenta@mixteco.utm.mx ";
       String RCPT_TO = "RCPT TO: destino@nuyoo.utm.mx ";
       String DATA = "DATA"; // inicio del mensage
       String ASUNTO = "Subject: Prueba Java\n";

       // Nota: "\n.\n" indica el final el mensaje
       String MENSAJE = "Cadena de mensaje\n.\n";

       Socket smtp = null;   // socket de SMTP

       try {  // Nota: 25 es el número de puerto SMTP por omisión
           smtp = new Socket("mixteco.utm.mx", 25);
           OutputStream os = smtp.getOutputStream();
           ps = new PrintStream(os);
           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 85. 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
8                               implements Runnable {
9     TextField id;
10    TextArea display;
11    Panel boardPanel, panel2;
12    Square board[][], currentSquare;
13    Socket connection;
14    DataInputStream input;
15    DataOutputStream output;
16    Thread outputThread;
17    char myMark;
18
19
20    public void init()
21    {
22       setLayout( new BorderLayout() );
23       display = new TextArea( 4, 30 );
24       display.setEditable( false );
25       add( "South", display );
26
27       boardPanel = new Panel();
28       boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );
29       board = new Square[ 3 ][ 3 ];
30
31       for ( int row = 0; row < board.length; row++ )
32          for (int col = 0; col < board[row].length; col++ ) {
33             board[ row ][ col ] = new Square();
34             boardPanel.add( board[ row ][ col ] );
35          }
36       id = new TextField();
37       id.setEditable( false );
38       add( "North", id );
39
40       panel2 = new Panel();
41       panel2.add( boardPanel );
42       add( "Center", panel2 );
43    }
44
```

```
45    public void start()
46    {
47       try {
48          connection =
49             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 86. Ejemplo - Cliente de TicTacToe.

26.2.2 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:

```java
//Ejemplo de un servidor genérico multihilado
import java.net.*;
import java.io.*;
import java.util.*;

public class GenericServer {
    // 1234 puede ser cualquier número de puerto que se determine
    int serverPort = 1234;
    public static void main(String args[]){
        //crear un objeto de servidor y ejecutarlo
        GenericServer server = new GenericServer();
        server.run();
    }
    public GenericServer() {
        super();
    }
    public void run() {
        try {
            //crear un socket servidor en el puerto especificado
            ServerSocket server = new ServerSocket(serverPort);
            do {
                //hacer un ciclo infinito para aceptar conexiones entrantes
                Socket client = server.accept();
                //crear un hilo nuevo para cada conexión
                (new ServerThread(client)).start();
            } while(true);
        } catch(IOException ex) {
            System.exit(0);
        }
    }
}

class ServerThread extends Thread {
    Socket client;
    //almacenar una referencia al socket en el que
    //está  conectado el cliente
    public ServerThread(Socket client) {
        this.client = client;
    }
    //este es el método inicial del hilo
    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 87. Ejemplo de un servidor genérico multihilado.

El programa servidor del gato, el cual lógicamente debe ejecutarse antes que los clientes para estar listo a recibir las conexiones. Es importante señalar que estos ejemplos no contienen más que una validación simple de las casillas ocupadas y el turno. No realiza por ejemplo, una validación para ver si alguno de los jugadores 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 88. Ejemplo - Servidor de TicTacToe.

Referencias

Apéndice A. Herramientas adicionales sugeridas

BlueJ

BlueJ [3] es un programa desarrollado por la universidades de *Kent* y *Deakin* para ayudar a los estudiantes a entender programación orientada a objetos en Java, particularmente ayuda a entender la herencia.

A partir de un diagrama de clases, BlueJ puede generar el código básico de la clase en Java, el cuál puede ser editado y compilado conforme las necesidades del programa. El programa es básico -ver figura **??** - y fácil de usar permitiendo entender estructuras complejas en las relaciones de herencia.

El código Java generado por BlueJ para el diagrama de la figura anterior es el siguiente:

```
1  /**
2   * Write a description of class Vehiculo here.
3   *
4   * @author (your name)
5   * @version (a version number or a date)
6   */
7  public class Vehiculo
8  {
9          // instance variables - replace the example below with your own
10         private int x;
11
12         /**
13          * Constructor for objects of class Vehiculo
14          */
15         public Vehiculo()
16         {
```

[3] Ver: http://www.bluej.org/

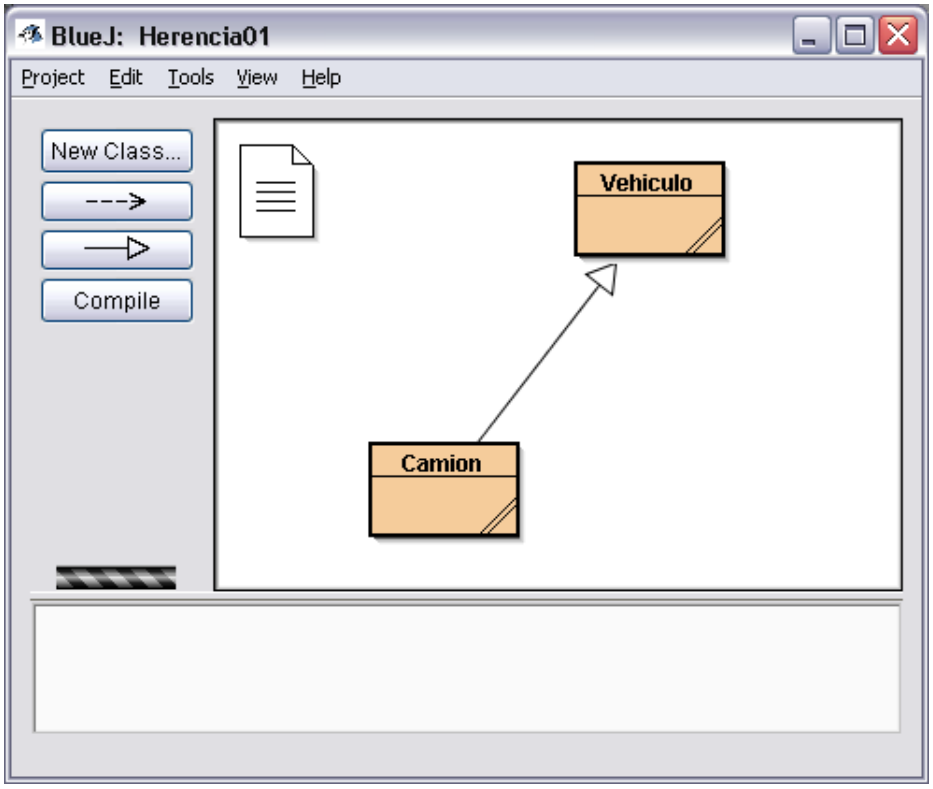


Fig. 26.1. Interface de BlueJ

```
17                  // initialise instance variables
18                  x = 0;
19          }
20
21          /**
22           * An example of a method - replace this comment with your own
23           *
24           * @param  y   a sample parameter for a method
25           * @return     the sum of x and y
26           */
27          public int sampleMethod(int y)
28          {
29                  // put your code here
30                  return x + y;
31          }
32  }
33
34
35
36  /**
37   * Write a description of class Camion here.
38   *
39   * @author (your name)
40   * @version (a version number or a date)
41   */
42  public class Camion extends Vehiculo
43  {
44          // instance variables - replace the example below with your own
45          private int x;
46
47          /**
48           * Constructor for objects of class Camion
49           */
50          public Camion()
51          {
52                  // initialise instance variables
53                  x = 0;
54          }
55
56          /**
57           * An example of a method - replace this comment with your own
58           *
```

```
59          * @param  y   a sample parameter for a method
60          * @return     the sum of x and y
61          */
62         public int sampleMethod(int y)
63         {
64                 // put your code here
65                 return x + y;
66         }
67 }
```

Listing 89. Ejemplo de código generado por BlueJ.