10

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

PRACTICAS DE LABORATORIO PARA INTRODUCCION A CIENCIAS DE LA COMPUTACION I (CON JAVA)

T E S I S

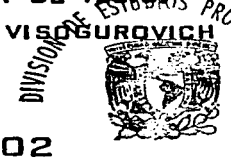
QUE PARA OBTENER EL TITULO DE LICENCIADO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A:

CANEK PELAEZ VALDES

FACULTAD DE CIENCIAS
UNAM

DIRECTORA DE TESIS:
M. EN C. ELISA VISCOVICH



TESIS CON
FALLA DE ORIGEN

2002

FACULTAD DE CIENCIAS
SECCION ESCOLAR

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

M. EN C. ELENA DE OTEYZA DE OTEYZA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

PRACTICAS DE LABORATORIO PARA INTRODUCCION A CIENCIAS DE LA COMPUTACION I (CON JAVA)

realizado por CANEK PELAEZ VALDES

con número de cuenta 9420313-4 , quién cubrió los créditos de la carrera de CIENCIAS DE LA COMPUTACION

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M. EN C. ELISA VISO GUROVICH
Propietario	M. EN C. JOSE DE JESUS GALAVIZ CASAS
Propietario	M. EN I. MARIA DE LUZ GASCA SOTO
Suplente	LIC. FRANCISCO SOLSONA CRUZ
Suplente	LIC. MANUEL SUGAWARA MURO

Consejo Departamental de MATEMATICAS

FACULTAD DE CIENCIAS

DRA. AMPARO LOPEZ GAONA

DEPARTAMENTO DE MATEMATICAS

Introducción a Ciencias de la Computación I

Manual de Prácticas de Laboratorio

Canek Peláez Valdés

A Gerardo Peláez Ramos; por la disciplina
y el flujo inagotable de sabiduría y conocimientos.

A María Eugenia Valdés Vega; por la alegría
y la forma de vida que quiero imitar.

A papá y mamá,
por todo lo demás.

Agradecimientos

Este trabajo es resultado, directa e indirectamente, de muchas personas además de mí.

En primer lugar quiero darle las gracias a Liliana; cuyo amor e inspiración han sido mi sustento durante mucho tiempo, y que nunca dejó de mostrar confianza hacia mí respecto este trabajo. Le agradezco infinitamente todo el amor, todo el cariño, todos los momentos, y que gracias a ella sé que el cielo existe y que está frente a nosotros, si sabemos mirar en la dirección correcta.

Quiero agradecer, por supuesto, a mis padres; por permitirme siempre pensar, sentir, y actuar como yo quisiera, y por apoyarme siempre a pesar de mis múltiples tropiezos y errores. En particular además quiero agradecer a papá, por tomarse la molestia de utilizar toda su experiencia en corrección de ortografía y estilo para ayudarme a mejorar este trabajo. A mamá Susi, claro, por darme el lujo de sólo tener que preocuparme por salir bien en la escuela. A mi hermano Amílcar, porque gracias a él muchas veces en mi vida supe qué decisiones debía tomar.

En la Facultad de Ciencias, mi hogar, están las personas que me acompañaron, que me motivaron y que me ayudaron para terminar este trabajo y a quienes les debo su sabiduría, su amistad y su apoyo.

A La Banda (así, con mayúsculas, como se escribe para quien no lo sepa), mis hermanos y hermanas, no sólo por todas las clases y todos los trabajos y todos los exámenes (no, perdón, los exámenes no son en equipo...), sino además por haber estado ahí, por acompañarme y apoyarme, no sólo en la fiesta y el relajo, ni sólo en la escuela y el trabajo; sino por de verdad haber sido hermanos, familia a la que quiero y siempre querré, y por haberme cuidado y haberse preocupado por mí en los momentos negros, los que en serio duelen. A ellos, a Liliana, Rafa, Erick, Oscar, Citlali, Karina, Yazmín, Gustavo, Edgar y Egar, les debo más de lo que puedo explicar, y mucho más de lo que jamás podré pagarles.

Mención aparte merecen mis hermanos, Juan José y Enrique, por esa camaradería que se da pocas veces en la vida, por todas las horas que hemos pasado platicando, todas las películas que hemos visto, y (por supuesto) todos los *frags* en Quake que inútilmente intentaron evitar que les hiciera.

Gracias en especial merece Verónica, amiga del alma en quien puedo decir que confié la vida, y que no me decepcionó jamás por ello. A ella le debo más de lo que jamás podrá nadie pagar, y además sé que tiene la grandeza suficiente para no cobrármelo.

Y por supuesto, a Ilse... donde quiera que esté.

La planta académica de la Facultad de Ciencias cambió mi vida. Me hizo abrir los ojos un nuevo mundo y a una forma de vida que espero pueda tener algún día.

Le agradezco a Miguel Lara, por explicarme lo que es ser alumno de Ciencias, y lo que es aprender matemáticas. A Pablo Barrera, por hacerme trabajar aunque yo no quisiera (y de verdad, no quería). A Mary Glazman (QEPD), por supuesto, por esos exámenes que odié. A

Jaqueline Cañetas, por sus fabulosas clases de Geometría. A Jefferson King, por las clases más divertidas de Cálculo, y por mostrarme que aprender (y enseñar), puede ser muy divertido. A Javier García (con todo y las clases a las 7:30 de la mañana). A Angélica, porque si no hubiera sido por ella, no paso proba (no hubiera entrado...) A Lucy Gazca, por $\log_2(n)$. A Favio Miranda, por las chelas en Querétaro, y por Análisis Lógico (¿era ése el orden?). A Roberto, por los 1,267 leds que hicimos explotar en su clase, uno detrás de otro, sólo por ver si siempre ocurría lo mismo... siempre ocurría. A José Galaviz, quien puede constatar que nunca en la vida he copiado o dejado que me copien. A Hanna Oktaba, por su respeto y sincera preocupación. Y a Francisco Raggi, por enseñarme Álgebra Moderna (y como tratar a una dama...).

Hay dos ayudantes que me dieron clase a los que tengo que dar gracias también. A Francisco Solsona, por iniciarme en este vicio que resultó ser Linux (aunque haya traicionado a la causa y comenzado a usar FreeBSD), y a Manuel Sugawara. No fue Manuel quien me enseño a programar; pero sí fue él quien soportó con paciencia infinita y estómago de hierro todas mis dudas y todas mis consultas a lo largo del semestre en que aprendí a programar.

Dos profesores más merecen especial mención. Sergio Rajsbaum, por la que fue tal vez la clase más fabulosa que jamás haya tomado, y por abrirme los ojos a lo que es realmente la computación teórica. Y Benjamín Macías, por varias de las clases más divertidas que tomé en la Facultad, por enseñarme a trabajar aunque nadie me lo exigiera, y por la preocupación personal en uno de los momentos más serios de mi vida. Estoy seguro de que no quiero ser exactamente como ninguno de ellos dos, pero estoy seguro de que lo que sea que quiero ser incluye muchas de sus cualidades. Y probablemente varias de las fallas.

Y ese es el mejor homenaje que puedo hacerle a alguien.

Por supuesto, falta gente. Mucha gente. Una disculpa y un abrazo a todos los que saben que los quiero y los estimo, y sólo espero que me perdonen que en este fin del camino se me haya olvidado dedicarles una línea siquiera. Sin embargo, sé que sabrán entender, y perdonar.

Sólo alguien más. Alguien a quien debo tanto que no hay palabras o gestos que puedan alcanzar para pagarle. Alguien a quien quiero como una madre (¡otra!), que tiene todo mi cariño, todo mi respeto, toda mi lealtad.

Hablo por supuesto de la profesora Elisa Viso Gurovich, asesora de este trabajo, quien durante toda esta carrera (desde su primer clase en mi primer semestre, hasta este ansiado momento), me brindó su apoyo, su sabiduría, su experiencia, sus consejos, su computadora, su café, su cubículo, su refrigeradorcito y su hornito de microondas (y faltaron las chelas).

Por ser maestra, amiga, apoyo, madre; alguien a quien siempre pude acudir y que siempre me recibió; por su cariño y sabiduría, su experiencia y su intuición; por su preocupación y su trabajo; por todo lo que no puedo expresar pero que ella y yo sabemos que le debo, te lo agradezco Elisa.

Gracias.

(...los chocolates idiota, siempre olvidas los chocolates...)

Introducción

A working program is one that has only unobserved bugs.

– Murphy's Laws of Computer Programming

En los últimos años, en los niveles de enseñanza superior en el área de computación, se ha optado de manera significativa en el primer curso de programación por enseñar el paradigma Orientado a Objetos, y en menor medida el Funcional.

Centrándonos en estos dos paradigmas (el Orientado a Objetos y el Funcional), los lenguajes representativos que más han logrado atraer la atención de educadores y estudiantes, y que más presentes están actualmente son Java y Scheme, respectivamente.

La discusión en cuanto a cuál de ellos debe ser el primer lenguaje de los estudiantes de computación ha alcanzado niveles de discusión religiosa, sin que ningún bando haya dado ningún argumento lo suficientemente aplastante para definir un ganador.

Lo más probable es que no exista tal ganador, y que ambos leguajes (y paradigmas) funcionen igual de bien para el propósito del primer curso de programación a nivel licenciatura: que la gente aprenda a programar. Por supuesto, *siempre y cuando exista un programa adecuado para el curso.*

Cuando nos referimos a "los niveles de enseñanza superior en el área de cómputo", esto abarca desde cursos que se dan a estudiantes de primer año de distintas carreras en algunas universidades estadounidenses, hasta los primeros cursos de programación que se dan en el MIT.

Para la elaboración de un programa de enseñanza adecuado, es necesario especificar a *qué tipo de personas* está dirigido el curso, y en *qué condiciones* se da.

Nosotros nos enfocamos en la Universidad Nacional Autónoma de México, en la Facultad de Ciencias, y enfocados principalmente en los alumnos de la carrera de Ciencias de la Computación que ahí se imparte.

Con este conjunto de estudiantes en mente, pensamos que el lenguaje a utilizarse debe ser Java, y que el curso debe estar fuertemente orientado a objetos, aunque también pensamos que el enfoque principal del paradigma funcional, la recursión, se debe introducir desde una etapa temprana.

Con el nivel de los estudiantes de la carrera en mente, y dada la forma en que actualmente se diseñan y programan las aplicaciones modernas, consideramos que se debe incluir en el curso temas como son el manejo de hilos de ejecución (*threads*) y la programación orientada a eventos.

Hay una serie de problemas que mucha gente le ve a Java (en particular) y al paradigma Orientado a Objetos (en general) para el primer lenguaje de programación:

- La sintaxis es demasiado complicada para un primer curso.

- El esquema *escribir* → *compilar* → *ejecutar* es mucho más oscuro y frustrante que el de *escribir* → *interpretar*.[1]
- Los alumnos pierden más el tiempo tratando de que su programa *compile* en lugar de preocuparse en *cómo* resolver bien el problema.
- Java utiliza excepciones y eventos de manera agresiva, los cuales son conceptos de niveles avanzados en programación.
- Java utiliza métodos de encapsulamiento muy diversos que pueden resultar confusos (**public, protected, private, package**).
- Java utiliza agresivamente los hilos de ejecución.

Hay que entender a qué tipo de estudiantes va dirigido el curso para entender nuestra decisión de utilizar Java.

El primer curso de programación de la carrera de Ciencias de la Computación (Introducción a Ciencias de la Computación I) se da en segundo semestre, lo que significa que los estudiantes a esa altura tienen los conocimientos matemáticos necesarios para un buen nivel de abstracción.

Esto significa, en general, que tienen conocimientos básicos de lógica de predicados, que son capaces de entender y realizar una demostración formal, y que tienen (o deberían tener) la madurez matemática mínima de la Facultad.

Además de este inicio en la formación científica, los alumnos de la carrera de Ciencias de la Computación (CC) *eligieron* la carrera. No se está lidiando con personas que aún no se deciden por alguna profesión y llevan una materia de computación para "ver si les gusta". Se está tratando con gente que de manera libre *eligió* la carrera y que espera dedicar el resto de su vida (o gran parte de ella) a la computación.[2]

Como la gente para la que está pensada la materia es gente que va a dedicarse a la computación, se puede responder puntualmente a las objeciones a Java como primer lenguaje:

- Ciertamente la sintaxis de Java (igual que la de los lenguajes estructurales e imperativos) es mucho más compleja que la de la mayoría de los lenguajes funcionales, en particular de Scheme. Sin embargo, la sintaxis es sólo cuestión de forma y está disponible siempre para consulta. Además, Java es muy consistente en cuanto a sintaxis se refiere, lo que implica que sólo hay que aprender algunos patrones sintácticos para llegar a comprenderla.

  Por otra parte, la sintaxis de Java ayuda mucho más a cursos subsecuentes de la carrera, como Introducción a Ciencias de la Computación II, donde se ven estructuras de datos.

- El saber cómo compilar un programa es parte fundamental de la enseñanza de un computólogo. La mayoría de los grandes sistemas hoy en día son compilados, por lo que no es sólo deseable sino necesaria la amplia experiencia con este tipo de lenguajes.

  Un buen compilador (como lo es el de Java) detecta todos los errores sintácticos, contrario a los que se marcan en el momento de interpretar, por lo que resulta más eficiente la depuración del programa.

[1] Casi todos los lenguajes interpretados de la actualidad compilan a *bytecode*; sin embargo, esto sigue siendo en la mayoría de los casos algo transparente al programador.

[2] El hecho de que esta elección haya sido o no acertada no es algo que le compete a los que imparten los cursos en la carrera; los cursos deben darse pensando en la gente que quiere tomarlos y que les sacará provecho. El caso en el que el estudiante eligió otra carrera y fue enviado a ésta es uno que debe ser resuelto por la universidad.

- Las excepciones fomentan buenos modales para programar. Ayudan a que un programa muera graciosamente, y facilitan el manejo de errores en un programa. El fomentar su uso debe ser parte integral de la enseñanza al programar.

  Y aunque apenas comienzan a aparecer cursos introductorios que incluyan eventos, lo cierto es que actualmente es casi imposible encontrar programas que no los usen, sobre todo por la presencia cada vez más intensa de Interfaces Gráficas de Usuario (IGUs), y debe por lo menos mencionarse su existencia y explicarse su funcionamiento.

- Los métodos de encapsulamiento (los de Java en particular y los de Orientación a Objetos en general), son otra buena costumbre de programación. Ayuda a la modularización y reutilización de código, y facilita la explicación del uso y creación de bibliotecas.

- Al igual que con los eventos, actualmente es casi imposible encontrar aplicaciones que no utilicen hilos de ejecución, y es necesario que los alumnos comprendan que tendrán que lidiar con ellos si quieren hacer algo medianamente serio (como eran los apuntadores hace apenas unos años).

Por supuesto, además de todo esto Java incluye muchos atractivos por sí mismo:

- Es un lenguaje nuevo y que tiene un diseño cuidadoso y bien fundamentado.

- Tiene recolector de basura, lo que permite al programador el no preocuparse por liberar memoria, ya que la Máquina Virtual lo hace automáticamente.

- Es un lenguaje utilizado en el mundo real (no es un lenguaje "académico") y que actualmente está en auge.

- Tiene una de las bibliotecas más grandes (y útiles) en existencia.

Aun así, los argumentos sobre su complejidad y tamaño no pueden ser ignorados. Por eso en este trabajo suministramos a los alumnos una serie de bibliotecas especializadas (cajas negras) que se les proporciona para que comiencen a utilizar sin contratiempos ciertas características del lenguaje que puedan resultar complicadas, por ejemplo entrada y salida de datos en programa, ya sea interactuando con el usuario o con dispositivos de la computadora.

Las cajas negras son una abstracción de partes del lenguaje de programación Java que pueden resultar difíciles o confusas para un estudiante de segundo semestre de la carrera y que nunca ha programado.

Aspectos como eventos, interfaces gráficas, y algunas estructuras de datos se encapsularon en clases opacas que los estudiantes aprenderán a utilizar primero, para después implementar sus propias versiones, y por último abrir las cajas (ver cómo están hechas) y comparar implementaciones. En particular, se evita que los estudiantes tengan que atrapar excepciones sin saber por qué lo hacen, ya que para entender propiamente el concepto de excepción, necesitan saber herencia.

Una característica importante es que no se les dará a los estudiantes un subconjunto o una versión ligera del lenguaje Java. Sencillamente en un principio se les facilitará una abstracción de algunos aspectos para que después ellos implementen una abstracción similar.

El enfoque de este curso está fundamentado en la idea de que los alumnos deben de estar en frente de una computadora un tiempo considerable del semestre. Creemos firmemente (por

experiencia empírica) que a los alumnos se les facilita más el comprender los conceptos básicos de programación si pueden poner en práctica de inmediato lo que ven en clase. Desde nuestro punto de vista, la parte práctica o técnica del curso es igual de importante que la teórica, y debe hacer ver a los alumnos que por bien hecho que esté un diseño, deben aterrizarlo en una buena implementación. Y eso no es posible si no tienen prácticas regulares a lo largo del curso. Por eso mismo, creemos también que es un requisito mínimo indispensable que el profesor del curso maneje el lenguaje de programación que se está dando, y que no se limite a ver los aspectos teóricos de la Orientación a Objetos.

La intención de este trabajo es justamente el proveer el material necesario para un primer curso de programación, teórico y práctico, que utilice Java como primer lenguaje y que esté orientado a los estudiantes de Ciencias de la Computación de la Facultad de Ciencias; y aunque ése es el principal conjunto de estudiantes a los que está dirigido el curso, estamos seguros de que las prácticas pueden adaptarse a cursos de programación de otras carreras y facultades, si el profesor está consciente de las diferencias que existan entre los alumnos a los que está dirigido este trabajo, y a los que piense darles el curso.

En total son doce prácticas de laboratorio (una de ellas opcional), que se reparten en las dieciséis semanas que dura el curso:

1. La práctica 1 se realiza en la segunda semana, y es una introducción sencilla al uso del compilador y del ambiente de programación.
2. La práctica 2 está pensada para la tercera semana del curso, y sirve para que el alumno vea cómo declarar e instanciar objetos, además de cómo mandar llamar métodos.
3. La práctica 3 se deja durante la cuarta semana, y en ella se ven tipos básicos, variables locales y operadores.
4. En la práctica 4, que se realiza en la quinta semana, se ven variables de clase y métodos, con lo que las clases quedan cubiertas. En esta práctica los estudiantes tienen dos semanas para resolverla.
5. La práctica 5 está planeada para la séptima semana del curso, y en ella se ven estructuras de control y listas.
6. La práctica 6 es para la octava seamana, y se ve herencia en ella.
7. La práctica 7 se deja en la novena semana, y se cubre entrada/salida y arreglos. En esta práctica los alumnos tienen dos semanas para concluirla.
8. La práctica 8 se ve en la décimo primera semana del curso, y en ella se estudia recursión.
9. En la práctica 9 se ven excepciones, y se deja en la décimo segunda semana.
10. En la práctica 10, que se deja en la décimo tercera semana del curso, se cubren interfaces gráficas. En esta práctica los alumnos tienen dos semanas para completarla.
11. La práctica 11 es para analizar paquetes y *jarfiles*. Se deja en la semana décimo quinta.
12. La práctica 12 se ve en la décimo sexta semana, y en ella se ven hilos de ejecución y enchufes.

Convenciones

Se ha tratado de utilizar una serie de convenciones coherentes para poder distinguir los diferentes contextos en los que aparece código en el texto.

Todo el código fue insertado en el *texto* utilizando el paquete listings, de Carsten Heinz, a menos que el paquete fuera incapaz de manejar el contexto. De cualquier forma, si no se podía usar el paquete listings, se trató de emular su funcionamiento.

El código que aparece dentro de bloques con líneas numeradas es código que está pensado para que se escriba directamente en un archivo o que está sacado de algún archivo ya existente. Por ejemplo

```
5  public class UsoMatriz2x2 {
6
7      public static void main (String[] args) {
8          Consola c;
9          c = new Consola ("Matrices");
10         ...
```

El código que aparece en cajas y sin numeración de líneas es código pensado para ejemplos cortos, no necesariamente existente o atado a alguna función en particular. Por ejemplo

```
c.imprimeln ("hola mundo");
```

Los comandos pensados para que el alumno los teclee en su intérprete de comandos están escritos utilizando el tipo `typewriter`, y se utiliza el carácter # para representar el *prompt* del intérprete. Por ejemplo

```
# javac -classpath interfaz1.jar:. Matriz2x2.java
```

Cada vez que aparece un nuevo concepto en el texto, se resalta utilizando el tipo *slanted*. Por ejemplo: "Generalmente diremos que la *clase principal* o *clase de uso* es la clase que mandamos ejecutar desde el shell con `java`."

Cuando un nombre o nombres aparezca entre < y >, significa que puede ser reemplazado por cadenas proporcionadas por el alumno. Por ejemplo

```
# java <NombreDeClase>
```

significa que `<NombreDeClase>` puede ser reemplazado por cualquier nombre de clase. También significará que puede ser reemplazado por más de un término, si el contexto lo permite. Por ejemplo en

```
# javac <VariosArchivosDeClases>
```

<VariosArchivosDeClases> podrá ser reemplazado por varios archivos de clases.

Todas las prácticas están precedidas por unos cuantos párrafos dirigidos al profesor del curso, con la idea de explicar el porqué de la práctica, qué tiempo se recomienda dar para que sea entregada, qué temas deben haber visto, y en general una justificación teórica de la práctica.

Figuras

El compilador de Java

Every non-trivial program has at least one bug

- Corollary 1 - A sufficient condition for program triviality is that it have no bugs.
- Corollary 2 - At least one bug will be observed after the author leaves the organization.

– Murphy's Laws of Computer Programming #1

Semana:	Segunda semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Entorno de trabajo; compilación/interpretación; errores semánticos, errores sintácticos.
Objetivo:	Que los alumnos utilicen los mensajes de error del compilador de Java para corregir y compilar satisfactoriamente una clase dada.

La primera práctica debe realizarse durante las primeras dos semanas del curso. Los alumnos deben saber desenvolverse en el ambiente de programación que se esté utilizando; deben saber crear y borrar directorios, y mover archivos entre ellos, así como ejecutar programas.

El profesor del curso debe haber acabado cualquier introducción de las ciencias de la computación que haya planeado, y los alumnos deben llegar a esta primera práctica sabiendo las características generales de Java, y haber visto en la teoría qué es compilar, ejecutar e interpretar, así como saber la diferencia entre errores sintácticos y semánticos.

Esta primera práctica debería llevarles a lo más dos horas, ya que el mismo compilador de Java explica dónde se localizan los errores y de qué tipo son. Sin embargo, para homogeneizar el resto de las prácticas, se recomienda que se les dé una semana para entregarla.

La idea general de esta práctica es enfrentar a los alumnos del curso con lo que será su ambiente de trabajo a lo largo del semestre: el sistema operativo, el editor de texto, y el compilador (sean cuales sean). Se les proporciona una clase de Java con algunos errores sintácticos, que deben corregir para poder compilarla y ejecutarla.

No se ve aún nada de Java propiamente; empero, durante la semana que tendrán para entregar la práctica, sería recomendable que el profesor comenzara a ver orientación a objetos y cómo la maneja Java; de esta manera los alumnos comenzarán a comprender cómo funciona la clase UsoReloj.

Práctica 1
El compilador de Java

1.1. Meta

Que el alumno aprenda a usar el compilador de Java para detectar errores de sintaxis y semánticos, y a generar *bytecode* ejecutable.

1.2. Objetivos

Al finalizar la práctica el alumno será capaz de:

- invocar al compilador de Java,
- interpretar los mensajes de error del mismo, y utilizarlos para corregir su programa,
- generar *bytecode* ejecutable, y
- ejecutar el programa compilado.

1.3. Desarrollo

Java es un lenguaje *compilado*, lo que quiere decir que un compilador se encarga de transformar las instrucciones de alto nivel de un programa, en código que la Máquina Virtual de Java (Java Virtual Machine o JVM) puede ejecutar.

Actividad 1.1 Invoca al compilador de Java sin ningún argumento, con la siguiente línea de comandos:

```
# javac
```

(Nota que sólo debes escribir `javac` y después teclear `Enter`; el # sólo representa el *prompt* de tu intérprete de comandos).
Anota todas las opciones que se pueden pasar al compilador.

El compilador de Java no sólo genera el código ejecutable por la JVM, también detecta errores de sintaxis y semánticos, mostrando en qué línea ocurren.

Además, el compilador tiene lo que es conocido como *recuperación*; al encontrar el primer error no se detiene, trata de continuar tanto como sea posible y seguir encontrando errores. Esto último es importante porque le permite al programador corregir un mayor número de errores por cada compilación, en lugar de tener que compilar cada vez que quiere encontrar el siguiente error.

Actividad 1.2 Compila los archivos de esta práctica con la siguiente línea de comandos:

```
# javac Reloj.java GraficosDeReloj.java UsoReloj.java
```

¿Cuántos errores marca? ¿Los entiendes? Pregunta al ayudante el significado de cada error.

El compilador de Java muestra en qué línea de un programa ocurre el error (si encuentra alguno). Si se utiliza un editor como XEmacs, y si está configurado de manera adecuada, se puede compilar dentro de XEmacs y saltar directamente a la línea en donde ocurre el error.

Actividad 1.3 Abre el archivo `UsoReloj.java` en XEmacs, y compila haciendo `C-x`, tecleando `javac UsoReloj.java` y dando `Enter` después.
Debe abrirse un segundo buffer en XEmacs donde se muestran los errores encontrados por el compilador. Cámbiate a ese buffer haciendo `C-x o` y teclea `Enter` en la primera línea que marque error. ¿Qué sucede?

Una vez que un programa está libre de errores, el compilador genera un archivo en *bytecode*.

La JVM ejecuta *bytecode*, un formato binario desarrollado para que los ejecutables de Java puedan ser utilizados en varias plataformas sin necesidad de recompilar.

El *bytecode* puede copiarse directamente a cualquier máquina que tenga una JVM, y ejecutarse sin cambios desde ahí. A eso se le conoce como *portabilidad a nivel binario*. Muchos otros lenguajes de programación tienen *portabilidad a nivel de código*.

1.4. Ejercicios

1. Corrige los errores que aparecen en la práctica, hasta que el programa compile y genere el *bytecode*. Utiliza los mensajes del compilador[1] para determinar la línea del error y en qué consiste. Todos los errores están en la clase UsoReloj, que está en el archivo `UsoReloj.java` (las clases de Java generalmente están en un archivo llamado como la clase, con extensión `.java`). *No* toques los otros dos archivos.

   (Puedes darle una mirada a los archivos `Reloj.java` y `GraficosDeReloj.java`; sin embargo, su funcionamiento no será explicado hasta prácticas posteriores).

[1] Sí, los mensajes están en inglés. Ni modo.

2. Una vez que el programa compile, ejecútalo con la siguiente línea de comandos:

```
# java UsoReloj
```

1.5. Preguntas

1. ¿Qué errores encontraste al compilar esta práctica? Explica en qué consisten.
2. Los errores que encontraste, ¿de qué tipo crees que sean, sintácticos o semánticos?
3. ¿Cuántos archivos en *bytecode* (los que tienen extensión `.class`) se generaron?

Usar y modificar clases

Bugs will appear in one part of a working program when another 'unrelated' part is modified.

– Murphy's Laws of Computer Programming #2

Semana:	Tercera semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Orientación a Objetos: clase, objetos, métodos, constructores.
Objetivo:	Hacer que el alumno conozca y utilice los conceptos de Orientación a Objetos, utilizando un enfoque más intuitivo que teórico.

Esta práctica se debe realizar durante la tercera semana del curso. El profesor debe haber visto ya el paradigma de Orientación a Objetos, en particular la definición de clase y objeto, con ejemplos, así como métodos y constructores. Los alumnos deben saber que a cada clase de Java le corresponde un archivo con extensión `.java`.

El alumno verá en esta práctica muchos conceptos que no serán explicados a nivel teórico todavía; la idea es que esté familiarizado con ellos para que en el momento en que sean abordados de manera formal, el estudiante tenga un antecedente intuitivo.

A pesar de que la práctica es relativamente larga y de que se abordan muchos conceptos (informalmente), los ejercicios son sencillos, limitándose a que los alumnos jueguen un poco con el código que se les ha dado, y a que se acostumbren al ciclo editar→compilar→corregir.

Esto significa que la explicación formal de muchas cosas se deja para prácticas futuras. Debe quedar claro, sin embargo, que todo lo que se menciona deberá ser explicado en detalle en un futuro cercano, aunque en esta práctica sólo se utilicen definiciones informales y se apele mucho a la intuición del alumno.

Esta manera de dar la segunda práctica del curso se debe a que en las siguientes prácticas los alumnos se verán enfrentados a ejercicios más complejos conceptualmente, y necesitarán varias herramientas que no es posible darles todavía (condicionales, operadores, bloques).

No importa cómo se dé el curso, es prácticamente imposible evitar estas dependencias cíclicas en los temas; para ver el tema B se necesita haber visto el tema A, pero para la cabal comprensión de éste, se necesitan elementos del tema B. Entonces lo que hacemos es dar una explicación intuitiva de las partes necesarias del tema B, lo suficiente para explicar el tema A. Una vez visto éste, podremos ver el tema B sin problemas.

El mejor ejemplo de estos casos es el **if**. No hay un solo algoritmo no trivial que se pueda dejar como ejercicio a los alumnos que no incluya al **if**[1]; y sin embargo, la manera correcta[2] de explicar el **if** es junto con **switch** y las iteraciones de Java. Una opción sería dejar algoritmos triviales hasta llegar a ese punto; pero entonces los alumnos se enfrentarían a una curva de aprendizaje más empinada (y con la sintaxis de Java tienen suficiente). Lo que hacemos entonces es explicar *intuitivamente* el **if**, dejando la explicación formal para prácticas posteriores.

Ya que los ejercicios son sencillos, también se recomienda dar sólo una semana a los alumnos para que entreguen esta práctica.

[1] Una manera ingeniosa de no utilizar **if** es usar la condicional aritmética ?:; sin embargo presenta el mismo problema de dependencias cíclicas. El operador ?: debe verse junto a los demás operadores de Java.

[2] Por supuesto, subjetivamente hablando.

Práctica 2
Usar y modificar clases

2.1. Meta

Que el alumno aprenda cómo funcionan las clases, cómo se instancian, qué son métodos y variables de clase, y cómo se llaman las funciones de una clase.

2.2. Objetivos

Al finalizar la práctica el alumno será capaz de:

- entender qué es una clase y declarar objetos de la misma,
- instanciar objetos de una clase con el operador **new**,
- entender qué son los métodos y variables de una clase,
- hacer que el objeto llame a las funciones de su clase con el operador . (punto),
- modificar las funciones de una clase para que sus objetos tengan un comportamiento distinto, y
- generar la documentación de una clase para saber qué funciones provee sin necesidad de leer el código directamente.

2.3. Desarrollo

En la práctica pasada, aprendimos cómo utilizar el compilador de Java para generar *bytecode* ejecutable, y ejecutamos un programa que mostraba un reloj en la pantalla.

En esta práctica analizaremos más detenidamente la clase UsoReloj para ver cómo funciona el programa, y modificaremos ciertas partes de la clase Reloj.

2.3.1. Las clases

Las clases de la práctica pasada son Reloj, GraficosDeReloj y UsoReloj.

Los objetos de la clase Reloj son como la maquinaria de un reloj de verdad. La maquinaria es la que se encarga de que el reloj funcione de cierta manera (de que avance o retroceda, etc.)

Los objetos de la clase GraficosDeReloj son como la parte externa de un reloj de verdad. Piensen en el Big Ben; los objetos de la clase GraficosDeReloj son la torre y las manecillas y la carátula con los números. En otras palabras, los objetos de la clase GraficosDeReloj no saben cómo funciona un reloj (ni tienen por qué saberlo). Sólo se encargan de mostrar la información del reloj al mundo exterior, poniendo las manecillas donde deben estar.

La clase UsoReloj es como el dueño de un reloj de verdad. Posee al reloj, lo pone a la hora requerida y en general hace con él todo lo que se puede hacer con un reloj.

La clase principal del programa es UsoReloj, en el sentido de que es en esta clase donde se utilizan las otras dos (Reloj y GraficosDeReloj). Generalmente diremos que la *clase principal* o *clase de uso* es la clase que mandamos ejecutar desde el intérprete de comandos con `java`.

La clase principal siempre tendrá el *punto de entrada* a nuestros programas; este punto de entrada es la función main.[1]

2.3.2. El método main

En la práctica anterior ejecutamos el programa con:

```
# java UsoReloj
```

Para ejecutar un programa en Java, se utiliza en general la siguiente línea de comandos:

```
# java <NombreDeUnaClase>
```

(Por su puesto, las clases del programa deben haberse compilado antes con `javac`).

Lo que hace entonces la JVM es ver el *bytecode* de la clase que es llamada, buscar la función main dentro de la clase, y ejecutarla. Si no existe un método main en esa clase, la JVM termina con un mensaje de error.

Por eso se le llama *punto de entrada* a la función main; es lo primero que se ejecuta en un programa en Java. Todas las clases en Java pueden tener su propio método main; sin embargo, en nuestras prácticas generalmente sólo una clase tendrá método main.

Actividad 2.1 Abre el archivo `UsoReloj.java` en XEmacs, y localiza la función main.

Noten que todas las partes de una clase de Java están formadas a partir de *bloques*, anidados unos dentro de otros, y todos adentro del bloque más grande, que es el de la clase misma. Un bloque comienza con un {, y termina con un }. Si descontamos el bloque de la clase, todos los bloques son de la forma:

```
{
    <expresión 1>;
    <expresión 2>;
    ...
    <expresión N>;
}
```

[1] Los términos *función*, *método*, y (aunque cada vez más en desuso) *procedimiento* serán considerados iguales en esta y las demás prácticas del curso. En Orientación a Objetos suele hacerse hincapié en que son *métodos*, no *funciones*, pero nosotros relajaremos esa regla ya que en Java sólo existen métodos.

Esto es un *bloque de ejecución*. En él, las expresiones del bloque se ejecutan en el orden en que aparecen. A un bloque también se les suele llamar *cuerpo*. Cada vez que un bloque comienza, la indentación del programa debe aumentar; esto facilita la lectura de código.

2.3.3. Instanciamiento de objetos

Ya dijimos que la clase UsoReloj es como el dueño de un reloj de verdad. Pero antes de poder usarse un reloj, debe adquirirse primero.

Para poder adquirir un objeto de la clase Reloj, se tiene que declarar. En la clase UsoReloj lo hacemos así:

```
14    Reloj r;
```

Así queda declarado un objeto de la clase Reloj, y el nombre con el que haremos referencia al objeto será r. Declarar un objeto es como cuando llenamos un catálogo para hacer compras por correo o por Internet; hacemos explícito qué queremos (en este caso un objeto de la clase Reloj), pero aún no lo tenemos. Para poseerlo, necesitamos crearlo.

Para crear o *instanciar* un objeto, se utiliza el operador new. En la clase UsoReloj lo hacemos de la siguiente manera:

```
17    r = new Reloj ();
```

El operador new llama al constructor de la clase Reloj. Veremos con más detalle a los constructores en la siguiente práctica.

Es hasta el momento de instanciar el objeto, cuando éste comienza a existir propiamente; ahora podemos comenzar a utilizarlo.

Actividad 2.2 En el archivo `UsoReloj.java`, en su función main, localiza dónde se declara al objeto r de la clase Reloj y también dónde se instancia.

Dentro de la función main de la clase UsoReloj ya tenemos un objeto de la clase Reloj que se llama r. Pero dijimos que los objetos de la clase Reloj son como la maquinaria de un reloj. Necesitamos la caja, la parte externa.

Para poner a la maquinaria la carátula y las manecillas, necesitamos un objeto de la clase GraficosDeReloj.

De igual manera, necesitamos declarar al objeto e instanciarlo. En la clase UsoReloj hacemos esto:

```
GraficosDeReloj rep;
...
rep = new GraficosDeReloj (r);
```

Y de esta manera ya tenemos un objeto de la clase GraficosDeReloj llamado rep.[2]

[2]Pueden declarar e instanciar objetos al mismo tiempo, utilizando GraficosDeReloj rep = new GraficosDeReloj (r);, pero por el momento no utilizaremos esta forma.

Noten que el constructor de la clase GraficosDeReloj no es igual al de la clase Reloj. El constructor de la clase Reloj no tenía parámetros (r = new Reloj()), mientras que el constructor de la clase GraficosDeReloj tiene un parámetro (rep = new GraficosDeReloj (r)). El parámetro es r; veremos con más detalle los parámetros en las siguientes prácticas.

La instanciación se puede leer: al instanciar un objeto de la clase GraficosDeReloj *le pasamos* como parámetro un objeto de la clase Reloj. Esto es como si atornilláramos la maquinaria del reloj a la caja; hacemos que la representación del reloj esté atada a su funcionamiento interno (y tiene sentido que sea en ese orden, ya que la caja de un reloj no tiene razón de ser sin el mecanismo).[3]

2.3.4. Usar métodos

Los *métodos* o funciones de una clase son los servicios a los cuales un objeto de esa clase puede tener acceso. En la clase UsoReloj, el objeto r de la clase Reloj sólo utiliza cuatro métodos: avanzaSegundo, avanzaMinuto, avanzaHora y defineTiempo.

Para llamar a una función utilizamos el operador . (punto). En la clase UsoReloj, llamamos a avanzaSegundo de la siguiente manera:

```
22    r.avanzaSegundo ();
```

En general, para llamar una función se utiliza:

```
<nombreObjeto>.<nombreFunción> (<parámetro_1>,...,<parámetro_N>);
```

Lo que hace entonces el operador . (punto) es ver de qué clase es el objeto (un objeto siempre sabe de qué clase es instancia), buscar la función por su nombre, y ejecutarla pasándole los parámetros.

Actividad 2.3 En el archivo `UsoReloj.java`, busca dónde se llama a las funciones avanzaSegundo, avanzaMinuto y avanzaHora y defineTiempo de la clase Reloj.

Estos métodos hacen lo que su nombre indica; avanzaSegundo hace que el segundero se mueva hacia adelante un segundo. Las otras dos (avanzaMinuto y avanzaHora) hacen lo mismo con el minutero y las horas respectivamente.

El método defineTiempo cambia el valor del segundero, del minutero y de las horas de un solo golpe. Recibe tres enteros como parámetro, así que para poner el reloj a las 7 con 22 minutos y 34 segundos tenemos que hacer esto:

---

[3] Aquí algunos de ustedes estarán pensando que no tiene mucho sentido separar la maquinaria de la caja; después de todo, cuando uno compra un reloj lo hace con la maquinaria y la caja juntas (aunque muchos relojes finos vienen con varios extensibles intercambiables, por ejemplo).

Esta es una de las características del paradigma Orientado a Objetos. Al tener separadas la parte que muestra al reloj, de la parte que maneja su funcionamiento, es posible cambiar la representación del reloj sin tener que reescribir todo el programa (podríamos escribir una clase llamada GraficosDeRelojDigital, que en lugar de usar manecillas utilizara el formato HH:MM:SS para representar el reloj, y ya no tenemos que preocuparnos del funcionamiento interno del reloj, que lo maneja la clase Reloj).

```
r.defineTiempo (7,22,34);
```

Ahora, una de las desventajas de separar el funcionamiento del reloj de su representación, es que esta última no cambia automáticamente cuando el estado del reloj cambia. Así que explícitamente tenemos que decirle a la representación (a la caja) que se actualice. Para esto, la clase GraficosDeReloj nos ofrece el método que se llama actualiza, que no recibe parámetros:

```
23    rep.actualiza ();
```

En la función main de la clase UsoReloj, utilizamos también la función espera de la clase GraficosDeReloj. Esta función sólo espera un determinado número de segundos (que es el parámetro que recibe) antes de que nada más ocurra.

Actividad 2.4 Abre el archivo `Reloj.java` y busca la definición de las funciones utilizadas en `UsoReloj.java`.
Si estás utilizando XEmacs como editor, puedes abrir el archivo y hacer C-s para comenzar una búsqueda interactiva. Esto quiere decir que mientras vayas tecleando, XEmacs irá buscando en el archivo lo que vayas escribiendo.
Pruébalo; abre el archivo `Reloj.java`, haz C-s y teclea avanzahora (así, todas minúsculas). Para cuando hayas terminado, XEmacs estará marcando la primera ocurrencia de la cadena `avanzaHora` en el archivo (observa que XEmacs ignora la diferencia entre mayúsculas y minúsculas).
Esa primera ocurrencia no es todavía la definición del método avanzaHora; si vuelves a dar C-s, XEmacs marcará ahora sí la definición que buscas. Utiliza C-s cuando quieras buscar algo rápido en XEmacs.

2.3.5. Variables locales y de clase

En la función main de la clase UsoReloj, declaramos dos variables, r y rep; la primera para *referirnos* a nuestro objeto de la clase Reloj, y la segunda para referirnos a nuestro objeto de la clase GraficosDeReloj.

En la clase Reloj, puedes ver al inicio del archivo las siguientes líneas:

```
20      private int horas;
21      private int minutos;
22      private int segundos;
```

También horas, minutos y segundos son variables. ¿Qué diferencia tienen de r y rep?

La primera diferencia es obvia; la declaración de horas, minutos y segundos está precedida por un private. La palabra clave private es un *modificador de acceso*; los veremos con más detalle en la práctica 3.

La segunda diferencia también es muy fácil de distinguir: r y rep están declaradas *dentro de una función* (main en este caso), mientras que horas, minutos y segundos están declaradas *dentro de la clase.*

La diferencia es importante; las variables declaradas dentro de una función son variables *locales*, mientras que las variables declaradas dentro de una clase son variables *de clase* o variables *miembro*.

Desde que vieron la declaración de horas, minutos y segundos, debió ser obvio que sirven para que guardemos los valores del mismo nombre de nuestros objetos de la clase Reloj.

Las variables de clase sirven justamente para eso; para guardar el estado de un objeto. Al valor que tienen en un instante dado las variables de clase de un objeto se le llama el *estado de un objeto*.

Veremos más acerca de las variables en las siguientes prácticas.

2.3.6. Métodos desde adentro

Ya vimos algunos métodos de la clase Reloj que se utilizan en la clase UsoReloj. ¿Para qué sirven los métodos?

Dijimos que las variables de clase guardan el estado de un objeto. La misión de los métodos de una clase es justamente cambiar ese estado, o para obtener información acerca de él de alguna manera. Por ejemplo, las funciones avanzaSegundo, avanzaMinuto y avanzaHora cambian el estado de nuestro objeto r, y la función actualiza cambia el estado de nuestro objeto rep, para que mueva nuestras manecillas a la hora actual.[4]

Veamos el método avanzaHora de la clase Reloj (recuerda que está en el archivo `Reloj.java`):

```
151   public void avanzaHora () {
152       horas++;
153       if (horas > 11) {
154           horas = 0;
155       }
156   }
```

¿Qué hace el método? En primer lugar, no recibe ningún parámetro. Esto tiene sentido, ya que la función siempre hará lo mismo (avanzar una hora), y por lo tanto no necesita ninguna información externa para hacerlo. Después, está la línea:

```
152       horas++
```

El operador ++ le suma un 1 a una variable. Veremos éste y más operadores en las siguientes prácticas.

Sumarle un 1 a horas es justamente lo que queremos (que las horas avancen en uno). Pero, ¿qué ocurre cuando son las 12 y le sumamos 1? En un reloj de verdad, la hora "da la vuelta" y se pone de nuevo en 1. En particular en nuestra clase Reloj estamos asumiendo que 12 es igual a cero (como ocurre con muchos relojes digitales), y entonces cuando horas vale 0, la manecilla de las horas apunta hacia el 12. Así que cuando horas valga 11 y se llame a avanzaHora, nuestro reloj deberá poner horas en 0. Eso es justamente lo que hacen las líneas

[4] Algunas veces también se utilizan funciones que no cambian el estado de un objeto, ni tampoco lo obtienen; el método espera es un ejemplo.

```
153         if (horas > 11) {
154             horas = 0;
155         }
```

La condicional if verifica lo que hay dentro de su paréntesis, y si es verdad, ejecuta el bloque que le sigue. Este if puede leerse "si horas es mayor que once, entonces haz que horas valga cero". Veremos con más detalle a if y otras condicionales más adelante.

Las demás funciones de la clase Reloj que hemos visto hasta ahora funcionan de manera similar a avanzaHora. Definiremos nuestras propias funciones en prácticas siguientes.

2.3.7. Comentarios dentro de los programas

En las clases Reloj y UsoReloj, habrás visto varias partes del código que empezaban con //, o que estaban encerradas entre /* y */. A estas partes se les llama *comentarios*, y sirven para que el programador haga anotaciones acerca de lo que está escribiendo. El compilador ignora los comentarios; sencillamente se los salta.

Los comentarios que empiezan con // son de una línea. El compilador ignora a partir de //, y lo sigue haciendo hasta que encuentra un salto de línea o que acaba el archivo, lo que ocurra primero.

Los comentarios que empiezan con /* son por bloque. El compilador ignora absolutamente todo lo que encuentra a partir de /*, y lo sigue haciendo hasta que encuentre un */. No se vale tener comentarios de este tipo anidados; esto es inválido:

```
/*
  /* Comentario inválido (está anidado). */
*/
```

También es inválido abrir un comentario /* y no cerrarlo nunca. Si el compilador ve un /*, y después llega al fin del archivo sin encontrar un */, entonces marcará un error.

2.3.8. JavaDoc

¿Cómo podemos saber qué métodos nos ofrece una clase?

La manera más sencilla es abrir el archivo de la clase y leer el código. Sin embargo, esto puede resultar tedioso, si la clase es larga y compleja.

Java ofrece un pequeño programa que nos permite observar los interiores de una clase de manera ordenada y elegante.

El programa se llama *JavaDoc*, y lee programas escritos en Java, con lo que genera páginas HTML que podemos ver utilizando cualquier navegador, como Mozilla o el Internet Explorer.

Actividad 2.5 Invoca a JavaDoc sin parámetros con la siguiente línea:

```
# javadoc
```

Anota todas las opciones que podemos utilizar.

Como puedes leer por lo que escribe en pantalla, el programa `javadoc` puede recibir como parámetros varios nombres de clases, o los archivos donde están escritas, y un directorio en donde poner la documentación generada.

Actividad 2.6 En el directorio donde están los archivos de la práctica, crea un directorio llamado `doc`, y ejecuta `javadoc` de la siguiente manera:

```
# mkdir doc/
# javadoc -d doc/ Reloj.java GraficosDeReloj.java
```

Pregúntale a tu ayudante cómo puedes utilizar Netscape o Mozilla para ver la documentación generada.

La documentación generada muestra, entre otras cosas, los métodos y constructores de la clase, incluyendo los parámetros que reciben.

Más adelante veremos cómo utilizar los comentarios de Java para ayudar a JavaDoc a generar información.

2.4. Ejercicios

1. Modifica la función avanzaMinuto de la clase Reloj para que cada vez que sea llamada, en lugar de avanzar un minuto avance diez. Recuerda que la clase Reloj está en el archivo `Reloj.java`.

   Compila y ejecuta de nuevo el programa para ver si tus cambios son correctos.

2. Con la documentación que generaste, lee qué parámetros recibe el segundo constructor de la clase GraficosDeReloj, y utilízalo en el método main de la clase UsoReloj, para que al ejecutar el programa la ventana del reloj sea más grande o pequeña. Juega con varios valores de x y y, compilando y ejecutando cada vez para comprobar tus resultados.

Nota que en el primer ejercicio sólo debes modificar el archivo `Reloj.java`, y en el segundo ejercicio sólo debes modificar `UsoReloj.java`. No es necesario que toques, y ni siquiera que mires `GraficosDeReloj.java`.

Los interiores de `GraficosDeReloj.java` los veremos más adelante, cuando hayamos manejado ya interfaces gráficas.

2.5. Preguntas

1. ¿Entiendes cómo funciona el método avanzaHora?
2. ¿Entiendes cómo funciona el método defineTiempo?
3. ¿Entiendes cómo funciona el método defineTiempoSistema? Si no, ¿qué partes no entiendes? Si lo entiendes, explícalo.

Variables, tipos y operadores

The subtlest bugs cause the greatest damage and problems.

- Corollary - A subtle bug will modify storage thereby masquerading as some other problem.

– Murphy's Laws of Computer Programming #3

Semana:	Cuarta semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Expresiones, tipos, operadores, variables referencias.
Objetivo:	Analizar las cosas a nivel de expresiones: tipos, operadores, variables y literales.

Esta práctica se realiza durante la cuarta semana del curso. El profesor debe ya haber visto tipos, operadores, expresiones y variables. Sería muy recomendable que para esta práctica se haya estudiado ya cómo se manejan los enteros en complemento a dos y el punto flotante, además de conversiones binario↔decimal, decimal↔binario, y binario↔hexadecimal, hexadecimal↔binario. El concepto de referencia puede darse durante el tiempo en que los alumnos realicen la práctica, ya que aunque se explica en qué consisten, los alumnos no las usarán hasta la siguiente práctica.

En esta práctica, el alumno verá los detalles del lenguaje Java a nivel de expresiones. La idea es por un momento bajarnos de nivel poniendo la Orientación a Objetos en un segundo plano (sin perderla de vista nunca) para analizar la sintaxis de Java en lo que se refiere a expresiones.

Si ponemos a los alumnos a escribir clases que hagan cosas interesantes (no triviales), la práctica crece considerablemente en tamaño, además de costarle más al alumno, ya que tiene que atacar y comprender dos conceptos nuevos al mismo tiempo: uno en alto nivel (las clases, como sus métodos determinan su comportamiento, como sus variables guardan su estado), y otro a bajo nivel (cómo sumar, cómo comparar, cómo hacer un OR lógico). Lo que hacemos entonces es darles todo lo de bajo nivel en una sola práctica, dejando los conceptos de alto nivel para la próxima.

La Orientación a Objetos no se olvida; sencillamente durante esta práctica no es el centro total de la atención. Sólo queremos que los alumnos manejen las herramientas básicas con las que sus clases funcionarán.

La práctica es larga, pero todos los conceptos que se abordan son sencillos, e intuitivos en su mayoría. Los ejercicios son fáciles también, ya que la idea sencillamente es que aprendan cómo sumar con Java (y las demás operaciones básicas).

Práctica 3
Variables, tipos, y operadores

3.1. Meta

Que el alumno aprenda a utilizar los tipos básicos de Java y sus operadores, y que entienda el concepto de referencia.

3.2. Objetivos

Al finalizar la práctica el alumno será capaz de:

- utilizar la clase Consola para hacer salida en pantalla,
- manejar variables locales, tipos básicos y literales de Java,
- entender lo que son referencias, y
- manejar operadores y expresiones.

3.3. Desarrollo

Un programa (en Java o cualquier lenguaje de programación), está escrito para resolver algún problema, realizando una o más tareas.

En el caso particular de Java, se utiliza la Orientación a Objetos para resolver estos problemas. Utilizar la Orientación a Objetos quiere decir, entre otras cosas, abstraer el mundo en objetos, y a estos objetos agruparlos en clases. En la práctica anterior, por ejemplo, utilizamos las clases Reloj y GraficosDeReloj, que abstraían el funcionamiento de un reloj y su representación externa.

Vamos a comenzar a resolver problemas utilizando Orientación a Objetos. Esto quedará dividido en dos partes fundamentales: la primera será identificar los objetos que representan nuestro problema, agruparlos en clases y definir precisamente el comportamiento (las funciones) que tendrán para que resuelvan nuestro problema.

La segunda parte es un poco más detallada. Una vez definido el comportamiento de una clase, hay que sentarse a escribirlo. Esto se traduce en implementar los métodos de la clase. Para implementarlos, es necesario manejar las partes básicas o atómicas del lenguaje: las variables, sus tipos, y sus operadores.

En esta práctica jugaremos con esta parte fundamental del lenguaje Java; en la siguiente utilizaremos lo que aprendamos aquí para comenzar a escribir nuestras clases.

3.3.1. Una clase de prueba

Para jugar con variables, vamos a necesitar un bloque de ejecución dónde ponerlas, o sea un método, y ya que necesitamos un método, vamos a necesitar también una clase dónde ponerlo.

Un problema es que no sabemos hacer todavía métodos, así que utilizaremos el método main, que ya utilizamos la clase pasada. Creemos pues una clase llamada Prueba, y utilicemos el método main de ella.

Nuestra clase Prueba quedaría como sigue:

```
1  public class Prueba {
2
3      public static void main (String[] args) {
4
5      }
6  }
```

Actividad 3.1 Crea un directorio llamado `practica3/`. Dentro de ese directorio, escribe la clase Prueba utilizando un archivo llamado `Prueba.java`.
Si estás usando XEmacs (y éste está bien configurado), y pones esto:

```
/* -*- java -*- */
```

en la primera línea, cada vez que abras el archivo, XEmacs hará disponibles varias funcionalidades especiales para editar archivos de Java. Entre otras cosas, coloreará de forma especial la sintaxis del programa.

Varios deben estar ahora haciéndose varias preguntas: ¿Qué quiere decir exactamente **public class Prueba**? ¿Para qué es el **static**? ¿De dónde salió el constructor de Prueba?

Todas esas preguntas las vamos a contestar en la siguiente práctica; en ésta nos vamos a centrar en variables, tipos y operadores.

La clase Prueba se ve un poco vacía; sin embargo, es una clase completamente funcional y puede ser compilada y ejecutada.

Actividad 3.2 Compila y ejecuta la clase Prueba con los siguientes comandos:

```
# javac Prueba.java
# java Prueba
```

No va a pasar nada interesante (de hecho no va a pasar nada), pero con esto puedes comprobar que es una clase válida.

Ahora, queremos jugar con variables, pero no va a servir de mucho si no vemos los resultados de lo que estamos haciendo. Para poder ver los resultados de lo que hagamos, vamos a utilizar a la clase Consola.

3.3.2. La clase Consola

Para tener acceso a la pantalla y ver los resultados de lo que le hagamos a nuestras variables, vamos a utilizar la clase Consola. Esta clase nos permitirá, entre otras cosas, escribir en la pantalla.

Actividad 3.3 Con la ayuda de tu ayudante, baja el archivo `interfaz1.jar`, y ponlo en el directorio `practica3/`.

¿Cómo vamos a utilizar la clase Consola dentro de nuestra clase Prueba? Primero, debemos decirle a nuestra clase acerca de la clase Consola. Esto lo logramos escribiendo

```
import icc1_1.interfaz.Consola;
```

antes de la declaración de nuestra clase. Veremos más detalladamente el significado de **import** cuando veamos paquetes.

En segundo lugar, cuando compilemos también tenemos que decirle al compilador dónde buscar a la clase Consola. Así que ahora compilaremos así:

```
# javac -classpath interfaz1.jar:. Prueba.java
```

Y en último lugar, cuando ejecutemos de nuevo tenemos que decirle a la JVM dónde está nuestra clase Consola:

```
# java -classpath interfaz1.jar:. Prueba
```

La opción `-classpath` del programa `java` la analizaremos también cuando veamos paquetes.

¿Qué podemos hacer con la clase Consola? De hecho, podemos hacer muchas cosas, que iremos descubriendo poco a poco conforme avancemos en el curso; pero por ahora, nos conformaremos con utilizarla para mostrar los valores de nuestras variables.

Actividad 3.4 Con la ayuda de tu ayudante, mira la documentación generada de la clase Consola. Se irán explicando y utilizando todos los métodos de la clase conforme se avance en el curso.

Para mostrar información con la consola, necesitamos crear un objeto de la clase consola y con él llamar a los métodos imprime o imprimeln. Por ejemplo, modifica el método main de la clase Prueba de la siguiente manera:[1]

[1]Observa que estamos utilizando ␣ para representar los espacios dentro de las cadenas.

```
1  import icc1_1.interfaz.Consola;
2
3  public class Prueba {
4
5      public static void main (String[] args) {
6          Consola c;
7          c = new Consola ("Valores de variables");
8          c.imprime ("Hola mundo.");
9      }
10 }
```

Compila y ejecuta la clase. El resultado lo puedes ver en la figura 3.1.

Figura 3.1: Consola

Para terminar el programa, cierra la ventana de la consola.
Analicemos las tres líneas que añadimos a la función main:

```
9   Consola c;
```

En esta línea sólo declaramos un nuevo objeto de la clase Consola llamado c.

```
10  c = new Consola ("Valores de variables");
```

En esta línea instanciamos a c. El constructor que usamos recibe como parámetro una cadena, que es el título de la ventana de nuestra consola. Al construir el objeto, la ventana con nuestra consola se abre automáticamente.

```
11  c.imprime ("Hola mundo.");
```

El método imprime de la clase Consola hace lo que su nombre indica; imprime en la consola la cadena que recibe como parámetro, que en nuestro ejemplo es "Hola mundo.".

Dijimos que podíamos utilizar los métodos imprime e imprimeln para mostrar información en la consola. Para ver la diferencia, imprime otra cadena en la consola:

```
9   Consola c;
10  c = new Consola ("Valores de variables");
```

```
11    c.imprime ("Hola mundo.");
12    c.imprime ("Adiós mundo.");
```

Compila y ejecuta la clase. El resultado lo puedes ver en la figura 3.2.

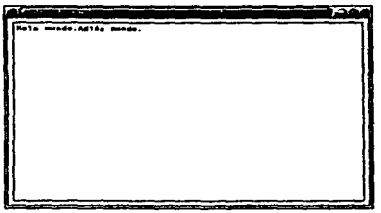


Figura 3.2: Resultado de imprime en la consola

Ahora, utilicemos el método imprimeln en lugar de imprime:

```
11    c.imprimeln ("Hola mundo.");
12    c.imprimeln ("Adiós mundo.");
```

Si compilas y ejecutas la clase, verás algo parecido a la figura 3.3.

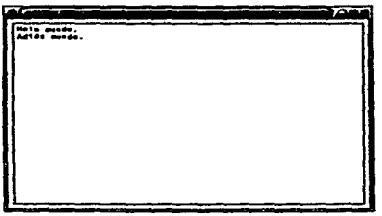


Figura 3.3: Resultado de imprimeln en la consola

La diferencia entre imprime e imprimeln es que la segunda imprime un salto de línea después de haber impreso la cadena que le pasamos como parámetro. El salto de línea en Java está representado por el carácter '\n'. Otra forma de conseguir el mismo resultado es hacer

```
11    c.imprime ("Hola mundo.\n");
12    c.imprime ("Adiós mundo.\n");
```

o incluso

```
c.imprime ("Hola mundo.\nAdiós mundo.\n");
```

La clase Consola ofrece muchos métodos, que iremos utilizando a lo largo de las demás prácticas. En esta sólo veremos imprime e imprimeln.

Actividad 3.5 Crea la clase Prueba en el archivo Prueba.java y realiza (valga la redundancia) pruebas con imprime e imprimeln.

3.3.3. Variables locales, tipos básicos y literales

Ya hemos trabajado con variables locales. En la práctica pasada, r y rep eran variables locales del método main en la clase UsoReloj. En esta práctica, c es un variable local de la función main en la clase Prueba.

Las variables locales se llaman así porque sólo son visibles *localmente*. Cuando escribamos funciones, no van a poder ver (y por lo tanto no podrán usar) a c porque esta variable sólo puede ser vista dentro de la función main. Se conoce como el *alcance* de una variable a los lugares dentro de un programa donde puede ser vista la variable.

Recordemos la declaración de r y rep de la práctica anterior:

```
Reloj r;
...
GraficosDeReloj rep;
```

Las dos son variables locales, pero tienen una diferencia muy importante: son de *tipos diferentes*. Java es un lenguaje de programación *fuertemente tipificado*, lo que significa que una variable tiene un único tipo durante toda su vida, y que por lo tanto nunca puede cambiar de tipo.

Esto quiere decir que lo siguiente sería inválido:

```
rep = new Reloj ();
```

Si intentan compilar algo así, javac se va a negar a compilar la clase; les va a decir que rep no es de tipo Reloj.

Vamos a empezar a declarar variables. Para declarar una variable, tenemos que poner el *tipo* de la variable, seguido del nombre de la variable:

```
int a;
```

Si queremos usar varias variables de un solo tipo, podemos declararlas todas juntas:

```
int a, b, c, d;
```

Con eso declaramos una variable de cierto tipo, pero, ¿qué son los tipos? Para empezar a ver los tipos de Java, declaremos un entero llamado a en el método main.

```
5       public static void main (String[] args) {
6           Consola c;
7           c = new Consola ("Valores de variables");
8
9           // Declaramos un entero "a";
10          int a;
11      }
```

La variable a es de tipo *entero* (int). Java tiene algunos tipos especiales que se llaman *tipos básicos*. Los tipos básicos difieren de todos los otros tipos en que ***NO*** son objetos; no tienen métodos ni variables de clase, y no tienen constructores. Para inicializar (que no es igual que instanciar) una variable de tipo básico, sencillamente se le asigna el valor correspondiente, por ejemplo:

```
a = 5;
```

Java tiene ocho tipos básicos, que puedes consultar en la figura 3.4.

Nombre	Descripción
byte	Enteros con signo de 8 bits en complemento a dos
short	Enteros con signo de 16 bits en complemento a dos
int	Enteros con signo de 32 bits en complemento a dos
long	Enteros con signo de 64 bits en complemento a dos
float	Punto flotante de 32 bits de precisión simple
double	Punto flotante de 64 bits de precisión doble
char	Carácter de 16 bits
boolean	Valor booleano (verdadero o falso)

Figura 3.4: Tipos básicos de Java

En el ejemplo de arriba, ¿qué representa el 5? El 5 representa una *literal*, en este caso de tipo entero. Las literales son valores que no están "atados" a una variable.

Todos los tipos básicos tienen literales:

```
int      entero;
double   doble;
char     caracter;
boolean  booleano;

entero    = 1234;
doble     = 13.745;
caracter  = 'a';
booleano  = true;
```

En general, a cualquier serie de números sin punto decimal Java lo considera una literal de tipo int, y a cualquier serie de números con punto decimal Java lo considera una literal de tipo double. Si se quiere una literal de tipo long se tiene que poner una L al final, como en 1234L, y

si se quiere un flotante hay que poner una F al final, como en 13.745F. No hay literales explícitas para byte ni para short.

Las únicas literales de tipo boolean son true y false. Las literales de carácter son 'a', 'Z', '+', etc. Para los caracteres como el salto de línea o el tabulador, las literales son especiales, como '\n' o '\t'. Si se necesita la literal del carácter \, se utiliza '\\'. Si se necesita un carácter internacional, se puede utilizar su código Unicode[2], por ejemplo '\u0041' representa el carácter 'A' (el entero que sigue a '\u' está escrito en hexadecimal, y deben escribirse los cuatro dígitos).

La fuerte tipificación de Java se aplica también a sus tipos básicos:

```
int a = 13;    // Código válido
int b = true; // ¡Código inválido!
```

Sin embargo, el siguiente es código válido también:

```
float x = 12;
byte  b = 5;
int   a = b;
```

Si somos precisos, la literal 12 es de tipo int (como dijimos arriba); y sin embargo se lo asignamos a una variable de tipo float. De igual forma, b es de tipo byte, pero le asignamos su valor a a, una variable de tipo int. Dejaremos la explicación formal de por qué esto no viola la fuerte tipificación hasta que veamos *conversión explícita de datos*; por ahora, basta con entender que un entero con signo que cabe en 8 bits (los de tipo byte), no tiene ninguna dificultad hacer que quepa en un entero con signo de 32 bits, y que todo entero con signo que quepa en 32 bits, puede ser representado por un punto flotante de 32 bits.

Por la misma razón, el siguiente código sí es inválido:

```
int  a = 1.2;
int  b = 5;
byte c = b;
```

El primero debe ser obvio; no podemos hacer que un punto flotante sea representado por un entero. Sin embargo, ¿por qué no podemos asignarle a c el valor b, cuando 5 sí cabe en un entero de 8 bits con signo? No podemos porque al compilador de Java *no le interesa* qué valor en particular tenga la variable b. Lo único que le importa es que es de tipo int, y un int *es muy probable* que no quepa en un byte. El compilador de Java juega siempre a la segura.

Actividad 3.6 Prueba los pequeños ejemplos que se han visto, tratando de compilar todos (utiliza el método main para tus ejemplos). Ve qué errores manda el compilador en cada caso, si es que manda.

---

[2]Unicode es un código de caracteres pensado para reemplazar a ASCII. ASCII sólo tenía (originalmente) 127 caracteres, y después fue extendido a 255. Eso basta para el alfabeto latino, con acentos incluidos, y algunos caracteres griegos; pero es completamente inútil para lenguas como la china o japonesa. Unicode tiene capacidad para 65,535 caracteres, lo que es suficiente para manejar estos lenguajes. Java es el primer lenguaje de programación en soportar nativamente Unicode.

¿Cómo vamos a imprimir en nuestra consola nuestros tipos básicos? Pues igual que nuestras cadenas:

```
int a = 5;
c.imprimeln(a);
```

Los métodos imprime e imprimeln de la clase Consola soportan ***todos*** los tipos de Java.

Una última observación respecto a variables locales (sean éstas de tipo básico u objetos). Es obvio que para usar una variable local en un método, tiene que estar declarada *antes de ser usada*. Esto es obvio por que no podemos usar algo antes de tenerlo. Empero, hay otra restricción: no podemos usar una variable antes de *inicializarla*; el siguiente código no compila:

```
int a;
int b = a; // No podemos usar la variable a todavía...
```

Estamos tratando de usar a sin que haya sido inicializada. Esto es inválido para el compilador de Java.

Dijimos que esto se aplica a tipos básicos y objetos, y más arriba dijimos que inicializar no era lo mismo que instanciar. Esto es porque también podemos inicializar variables que sean objetos sin necesariamente instanciar nada. Para entender esto, necesitamos comprender qué son las *referencias*.

3.3.4. Referencias

Cuando hacemos

```
Prueba p;
```

dijimos que estamos declarando un objeto de la clase Prueba. Así se acostumbra decir, y así lo diremos siempre a lo largo de estas prácticas. Sin embargo, formalmente hablando, estamos declarando una variable de tipo *referencia* a un objeto de la clase Prueba.

Una referencia es una dirección en memoria. Al instanciar p con **new**

```
p = new Prueba ();
```

lo que hace **new** es pedirle a la JVM que asigne memoria para poder guardar en ella al nuevo objeto de la clase Prueba. Entonces **new** regresa la dirección en memoria del nuevo objeto, y se guarda en la variable p.

Esto es importante; las variables de referencia a algún objeto tienen valores que son direcciones de memoria, así como las variables de tipo entero tienen valores que son enteros en complemento a dos, o las variables de tipo booleano tienen valores que son **true** o **false**.

Si después de instanciar p hacemos

```
Prueba p2;
p2 = p;
```

entonces ya no estamos *instanciando* a p2 (no se está asignando memoria a ningún nuevo objeto). Lo que estamos haciendo es *inicializar* a p2 con el valor de p, o sea, con la dirección

de memoria a la que hace referencia p. Después de eso, p y p2 son dos variables *distintas*, que tienen *el mismo* valor, o sea, que hacen referencia al mismo objeto.[3]

Todas las referencias (no importa de qué clase sean) tienen una única literal, que es **null**. Ésta es una referencia que por definición es inválida; no apunta nunca a ninguna dirección válida de la memoria. La referencia **null** es muy utilizada para varios propósitos, y en particular sirve para inicializar objetos que aún no deseamos instanciar. Veremos más ejemplos con **null** en las siguientes prácticas.

Una vez definidas las referencias, podemos dividir a las variables de Java en dos; toda variable de Java es de un tipo básico, o alguna referencia. No hay otras. Además de esto, todas las referencias apuntan a algún objeto, o tienen el valor **null**.

3.3.5. Operadores

Sabemos ya declarar todas las posibles variables de Java. Una vez que tengamos variables, podemos guardar valores en ellas con el operador de asignación =; pero además debemos poder hacer cosas interesantes con los valores que puede tomar una variable.

Para hacer cosas interesantes con nuestras variables, es necesario utilizar operadores. Los operadores de Java aparecen en la tabla 3.1.

Cuadro 3.1: Operadores de Java.

Operandos	Asociatividad	Símbolo	Descripción
posfijo unario	derecha	[]	arreglos
posfijo unario	derecha	.	selector de clase
posfijo n-ario	derecha	(<parámetros>)	lista de parámetros
posfijo unario	izquierda	<variable>++	auto post-incremento
posfijo unario	izquierda	<variable>--	auto post-decremento
unario prefijo	derecha	++<variable>	auto pre-incremento
unario prefijo	derecha	--<variable>	auto pre-decremento
unario prefijo	izquierda	+<expresión>	signo positivo
unario prefijo	izquierda	-<expresión>	signo negativo
unario prefijo	izquierda	~<expresión>	complemento en bits
unario prefijo	izquierda	!<expresión>	negación
unario prefijo	izquierda	new <constructor>	instanciador
unario prefijo	izquierda	(<tipo>)<expresión>	*casting*
binario infijo	izquierda	*	multiplicación
binario infijo	izquierda	/	división
binario infijo	izquierda	%	módulo
binario infijo	izquierda	+	suma
binario infijo	izquierda	-	resta
binario infijo	izquierda	<<	corrimiento de bits a la izquierda

[3]Se suele decir también que *apuntan* al mismo objeto.

Cuadro 3.1: Operadores de Java..... (continúa)

Operandos	Asociatividad	Símbolo	Descripción
binario infijo	izquierda	>>	corrimiento de bits a la derecha llenando con ceros
binario infijo	izquierda	>>>	corrimiento de bits a la derecha propagando signo
binario infijo	izquierda	<	relacional "menor que"
binario infijo	izquierda	<=	relacional "menor o igual que"
binario infijo	izquierda	>	relacional "mayor que"
binario infijo	izquierda	>=	relacional "mayor o igual que"
binario infijo	izquierda	instanceof	relacional "ejemplar de"
binario infijo	izquierda	==	relacional, igual a
binario infijo	izquierda	!=	relacional, distinto de
binario infijo	izquierda	&	AND de bits
binario infijo	izquierda	^	XOR de bits
binario infijo	izquierda	|	OR de bits
binario infijo	izquierda	&&	AND lógico
binario infijo	izquierda	||	OR lógico
ternario infijo	izquierda	<exp log > ? <exp> : <exp>	Condicional aritmética
binario infijo	derecha	=	asignación
binario infijo	derecha	+=	autosuma y asignación
binario infijo	derecha	-=	autoresta y asignación
binario infijo	derecha	*=	autoproducto y asignación
binario infijo	derecha	/=	autodivisión y asignación
binario infijo	derecha	%=	automódulo y asignación
binario infijo	derecha	>>=	autocorrimiento derecho y asignación
binario infijo	derecha	<<=	autocorrimiento izquierdo y asignación
binario infijo	derecha	>>>=	autocorrimiento derecho con propagación y asignación
binario infijo	derecha	&=	auto-AND de bits y asignación
binario infijo	derecha	^=	auto-XOR de bits y asignación
binario infijo	derecha	|=	auto-OR de bits y asignación

Los operadores en general se comportan como uno esperaría; si uno hace

```
int a = 3 + 2;
```

entonces a toma el valor 5. Todos los operadores tienen un dominio específico; cosas como

```
true + false;
```

no tienen sentido. Cuando se opera con tipos distintos, pero compatibles, se asume el tipo más general. Por ejemplo, si tenemos

```
int a = 2;
float x = 1.5;
```

entonces a+x es de tipo float.

Los operadores funcionan de acuerdo al tipo de los operandos. Si se dividen dos enteros con el operador /, se hace división entera; el resultado se trunca al mayor entero menor o igual al resultado de la división. Si se dividen flotantes o dobles, el resultado es la división real (o lo más cercano, de acuerdo a la precisión simple o doble de cada tipo).

Hay varios operadores en la tabla que funcionan sobre bits. Esto quiere decir que operan sobre un valor de tipo entero, pero fijándose en los bits, más que en el valor en sí.

Veamos un ejemplo

```
int a = 13;
int b = 19;
int c = a | b;
```

La variable a vale 13 y la b vale 19. En bits esto es

$$a = 00000000000000000000000000001101$$
$$b = 00000000000000000000000000010011$$

(recordemos que los enteros tienen 32 bits). Para no tener que escribir tanto, usamos hexadecimal para representar los valores y podemos verlos como 0x0000000D y 0x00000013.

¿Cuál es valor de la variable c entonces? Estamos usando el OR para bits; se aplica un OR a cada uno de los bits del entero, y el resultado es

$$c = 00000000000000000000000000011111$$

o bien 0x0000001F, que a su vez es el valor 31 en base 10. Siempre hay que tener en cuenta que los tipos enteros de Java (byte, short, int, long) tienen signo; eso quiere decir que el bit más significativo en estos tipos guarda el signo.

La tabla 3.1 está ordenada en orden de precedencia. La precedencia de los operadores determina el orden en que serán aplicados. Por ejemplo en

```
int a = 3 + 2 * 5;
```

el valor de a es 13, no 25, ya que la precedencia de la multiplicación es mayor que la de la suma. Si queremos que la suma se aplique antes, tenemos que hacer

```
int a = (3 + 2) * 5;
```

Pueden utilizar la tabla de operadores para saber qué operador tiene precedencia sobre cuáles; sin embargo, en general es bastante intuitivo. En caso de duda, poner paréntesis nunca hace daño (pero hace que el código se vea horrible; los paréntesis son malos^{MR}).[4]

Los operadores relacionales (<, >, <=, >=) funcionan sobre tipos numéricos, y regresan un valor booleano. La igualdad y la desigualdad (==, !=) funcionan sobre *todos* los tipos de Java. En el caso de las referencias, == regresará true si y sólo si las referencias apuntan a la misma posición en la memoria; al mismo objeto.

Hay que notar que la asignación (=) es muy diferente a la igualdad (==). La asignación toma el *valor* de la derecha (se hacen todos los cómputos que sean necesarios para obtener ese valor), y lo asigna en la *variable* de la izquierda. El valor debe ser de un tipo compatible al de la variable. También hay algunos operadores de conveniencia, los llamados "auto operadores". Son +=, −=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, y |=. La idea es que si tenemos una variable entera llamada a, entonces

```
a += 10;
```

es exactamente igual a

```
a = a + 10;
```

E igual con cada uno de los auto operadores. Como el caso de sumarle (o restarle) un uno a una variable es muy común en muchos algoritmos, los operadores ++ y −− hacen exactamente eso. Mas hay dos maneras de aplicarlos: prefijo y post fijo. Si tenemos

```
int a = 5;
int b = a++;
```

entonces a termina valiendo 6, pero b termina valiendo 5. Esto es porque el operador regresa el valor de la variable *antes* de sumarle un uno, y hasta después le suma el uno a la variable. En cambio en

```
int a = 5;
int b = ++a;
```

las dos variables terminan valiendo 6, porque el operador *primero* le suma un uno a a, y después regresa el valor de la variable.

Los operadores lógicos && (AND) y || (OR) funcionan como se espera que funcionen ($p \wedge q$ es verdadero si y sólo si p y q son verdaderos, y $p \vee q$ es verdadero si y sólo si p es verdadero o q es verdadero), con una pequeña diferencia: funcionan en *corto circuito*. Esto quiere decir que si hacemos

```
if (a < b && b < c)
```

y a < b es falso, entonces el && ya no comprueba su segundo operando. Ya sabe que el resultado va a dar falso, así que ya no se molesta en hacer los cálculos necesarios para ver si b < c es

[4] "Parenthesis are evil^{TM}".

verdadero o falso. Lo mismo pasa si el primer operando de un || resulta verdadero. Es importante tener en cuenta esto, porque aunque en Matemáticas Discretas nos dijeron que $p \wedge q \equiv q \wedge p$, a la hora de programar esto no es 100 % cierto.

Un operador bastante útil es la condicional aritmética (?:). Funciona de la siguiente manera:

```
int z = (a < b) ? 1 : 0;
```

Si a es menor que b, entonces el valor de z será 1. Si no, será 0. La condicional aritmética funciona como un if chiquito, y es muy útil y compacta. Por ejemplo, para obtener el valor absoluto de una variable (digamos a) sólo necesitamos hacer

```
int b = (a < 0) ? -a : a;
```

Hay varios operadores que funcionan con referencias. Algunos funcionan con referencias y con tipos básicos al mismo tiempo (como = y ==, por ejemplo); pero otros son exclusivos de las referencias.

Los operadores más importantes que tienen las referencias son new, que las inicializa, y . (punto), que nos permite interactuar con las partes de un objeto.

Hay otros operadores para los objetos, y otros para tipos básicos, que iremos estudiando en las prácticas que siguen.

3.3.6. Expresiones

Una vez que tenemos variables (tipos básicos u objetos), literales y operadores, obtenemos el siguiente nivel en las construcciones de Java. Al combinar una o más variables o literales con un operador, tenemos una *expresión*.

De hecho una variable o literal por sí misma es una expresión (trivial), pero como queremos hacer cosas interesantes, consideraremos expresiones a partir de la combinación de una o más variables o literales con uno o más operadores.

Las expresiones (como nosotros las consideraremos) no son atómicas; pueden partirse hasta que lleguemos a variables o literales y operadores. Podemos construir expresiones muy rápido:

```
a++;                                          // Esto es una expresión.
(a++);                                        // Esto también.
(a++)*3;                                      // También.
5+((a++)*3/2+5/r.dameMinuto()*r.dameHoras()); // Sí, también.
```

En general, todas las expresiones regresan un valor. No siempre es así, sin embargo, como veremos en la siguiente práctica.

Podemos hacer expresiones tan complejas como queramos (como muestra el último de nuestros ejemplos arriba), siempre y cuando los tipos de cada una de sus partes sean compatibles. Si no lo son, el compilador de Java se va a negar a compilar esa clase.

El siguiente nivel en las construcciones de Java lo mencionamos en la práctica pasada: son los bloques. La próxima práctica analizaremos nivel de construcción más importante de Java: las clases.

3.4. Ejercicios

Todas estas pruebas las tienes que realizar dentro del método main de tu clase Prueba.

1. Trata de obtener el módulo de un flotante.
2. Declara un **float** llamado x, y asígnale la literal 1F. Declara un **float** llamado y y asígnale la literal 0.00000001F. Declara un tercer **float** llamado z y asígnale el valor que resulta de restarle y a x. Imprime z en tu consola.
3. Imprime el valor de la siguiente expresión: 1>>1. Ahora imprime el valor de la siguiente expresión −1>>1.
4. Declara dos variables booleanas p y q.

   Asígnales a cada una un valor (**true** o **false**, tú decide), e imprime el valor de la expresión $\neg(p \wedge q)$ en sintaxis de Java. Después imprime el valor de la expresión $\neg p \vee \neg q$ utilizando la consola.

   Cambia los valores de p y q para hacer las cuatro combinaciones posibles, imprimiendo siempre el resultado de las dos expresiones. Con esto demostrarás DeMorgan caso por caso.

3.5. Preguntas

1. ¿Crees que sea posible asignar el valor de un flotante a un entero? ¿Cómo?
2. Mira el siguiente código

```
int a = 1;
int b = 2;
int c = 3;

(a > 3 && ++a <= 2) ? b++ : c--;
```

   Sin compilarlo, ¿cuál es el valor final de a, b, c? Compila y comprueba lo que pensaste con el resultado real.

Clases por dentro

A 'debugged' program that crashes will wipe out source files on storage devices when there is the least available backup.

– Murphy's Laws of Computer Programming #4

Semana:	Quinta semana.
Tiempo de entrega:	Dos semanas.
Prerrequisitos:	Restricciones de acceso, variables de clase, métodos, cadenas.
Objetivo:	Completar los conocimientos requeridos para que el alumno pueda escribir clases

Esta práctica se realiza durante la quinta semana del curso. El profesor debe haber visto variables de clase y métodos, junto con restricciones de acceso (público, privado, protegido, etc.), y las partes más importantes de los métodos (tipo de regreso, parámetros, diferencia entre declaración y uso).

Esta práctica tiene 21 páginas. Y se terminan de cubrir los conceptos que necesita un alumno para escribir una clase (variables de clase, métodos, restricciones de acceso, variables y métodos estáticos y finales).

No es por casualidad que las prácticas dos, tres y cuatro (ésta) hayan crecido tanto en tamaño y complejidad. Se está lidiando con el problema más importante que se genera al dar Java como primer lenguaje de programación: la propia sintaxis de Java.

Por eso se les da a los alumnos dos semanas para que entreguen esta práctica. Los ejercicios son mucho más complejos que los anteriores, aunque sólo tienen que escribir un par de métodos, que pueden hacerse con tres líneas de código cada uno. Pero involucran todos los conceptos que se han visto en las últimas tres prácticas.

Se les debe ayudar a los alumnos todo lo posible; tarea que principalmente tendrá el ayudante (al ser él el que está con ellos en los laboratorios), y debe prestarla y estar atento a quienes acaban los ejercicios, porque es muy probable que en cuando algún alumno encuentre la solución, se sienta tan emocionado que comience a pasarla a todos sus amigos (y éstos a su vez a todos sus amigos...) Hay que tratar de que cada quien haga su propia práctica;[5] quien no lo haga no tiene muchas posibilidades de hacer la siguiente.

Hay que hablar con el grupo y explicarles que no porque no encuentren la solución en la primera semana (pocos lo harán), significa que no sirven para la computación. Empero, esta

[5]Sabemos que es imposible evitar que los alumnos copien. Pero hay que tratar por todos los medios de explicarles que *tienen* que hacer esta práctica. Aunque pase lo que pase van a copiar de cualquier manera...

práctica comenzará a mostrar a quiénes se les va a hacer sencillo el curso y a quiénes se les va a dificultar (lo que no necesariamente determina quién va a salir bien y quien mal).

El tamaño (y complejidad) de las siguientes prácticas se volverá a reducir, y los alumnos que lleven a buen término esta práctica habrán sorteado la primera pendiente del curso, porque habrán comprendido las partes fundamentales de la sintaxis de Java.

Práctica 4
Clases por dentro

4.1. Meta

Que el alumno aprenda cómo escribir clases.

4.2. Objetivos

Al finalizar la práctica el alumno será capaz de:

- entender cómo está formada una clase,
- manejar cadenas, y
- escribir clases y clases de uso.

4.3. Desarrollo

En la práctica anterior, dijimos que usar el paradigma Orientado a Objetos consistía en abstraer en objetos el problema que queremos resolver. Abstraer en objetos significa identificar los elementos principales de nuestro problema, crear una clase que abarque todas sus instancias, y escribir los métodos necesarios de cada clase para que resuelvan nuestro problema.

¿Por qué decimos "crear una clase que abarque todas sus instancias"? Un problema, en ciencias de la computación, abarca muchos posibles (generalmente una infinidad) de casos concretos. No escribimos una solución para sumar 3 y 5; escribimos una solución para sumar dos números *a* y *b*, *para todos los posibles valores de a y b*.[1]

Ésa siempre ha sido la idea de las ciencias de la computación, desde el inicio cuando las computadoras utilizaban bulbos, y los programas se guardaban en tarjetas perforadas. La diferencia es *cómo* escribimos esta solución que abarque tantos casos como podamos.

Hace algunos años se ideó el paradigma Orientado a Objetos como un método para escribir las soluciones a nuestros problemas. No es el único paradigma; pero muchos pensamos que es el mejor y más general.

[1] A veces, sin embargo, resolver *todos* los casos es demasiado costoso, y restringimos el dominio de nuestro problema. Sin embargo, nunca escribimos una solución que sólo sirva con uno o dos casos concretos; siempre se trata de abarcar el mayor número de casos posibles.

La idea es que tomemos nuestro problema y definamos los componentes que lo forman. Cada uno de estos componentes será un objeto de nuestro problema. Pero no escribimos un solo objeto. Escribimos una clase que abarca a todas las encarnaciones posibles de nuestros objetos. Esto tiene muchísimas ventajas:

- Cada componente es una entidad independiente. Esto quiere decir que podemos modificar cada uno de ellos (hacerlo más rápido, más entendible, que abarque más casos), y los demás componentes no tendrán que ser modificados para ello. A esto se le llama *encapsulamiento*.

- Cuando resolvamos otro problema distinto, es posible que al definir sus componentes descubramos que uno de ellos ya lo habíamos definido para un problema anterior. Entonces sólo utilizamos esa clase que ya habíamos escrito. A esto se le llama *reutilización de código*.

  Tenemos un ejemplo de eso ya en las manos. La clase Consola fue escrita para resolver el problema de cómo mostrar información al usuario (también de cómo obtener información del usuario; veremos eso más adelante en esta práctica). El problema ya está resuelto; la clase Consola ya está escrita, y ahora la utilizaremos *casi todo* el curso, sin necesidad de modificarla o escribirla toda cada vez que queramos mostrar información al usuario.

Hay muchas más ventajas; pero para entenderlas necesitamos estudiar más conceptos. La ventaja más importante, sin embargo, es que nuestro problema puede quedar formalmente dividido en una serie de componentes bien definidos. Con esto *dividimos* al problema en problemas más pequeños. **Divide y vencerás.**

4.3.1. Las clases de Java

Venimos hablando de clases desde la primera práctica, y aún no hemos dicho bien qué es una clase. Una clase es un *prototipo* o *plantilla* que define los métodos y variables que son comunes a todos los objetos de cierto tipo. En la clase definimos el comportamiento de todos los objetos posibles de esa clase.

En Java las clases están compuestas de métodos y *variables de clase*. Las variables de clase no son iguales a las variables locales; tienen varias diferencias que iremos viendo a lo largo de esta práctica.

Las clases se declaran en Java como se muestra a continuación:

```
class <NombreDeClase> {

    <variableDeClase1>
    <variableDeClase2>
    ...
    <variableDeClaseM>

    <método1>
    <método2>
    ...
    <métodoN>
}
```

(Ya sabemos que todo esto debe estar en un archivo que se llame como la clase, o sea `<NombreDeClase>.java`).

Pusimos todas las variables de clase juntas al inicio, y todos los métodos juntos al final; pero realmente pueden ir en el orden que se quiera. Sin embargo, en estas prácticas será requisito que las variables de clase estén todas juntas al inicio.

Para aterrizar todo lo que vayamos discutiendo, iremos construyendo una clase para representar matrices de dos renglones por dos columnas que llamaremos (contrario a todo lo que podría pensarse) Matriz2x2. La primera parte ya sabemos hacerla:

```
1  class Matriz2x2 {
2  }
```

No sabemos hacer nada más, por lo que la dejaremos así por el momento. Sin embargo, la clase compila (no se puede ejecutar, porque no tiene método main).

Actividad 4.1 Comienza a escribir la clase Matriz2x2. Conforme se vayan añadiendo partes a lo largo de la práctica, agrégalas a la clase.

Veamos ahora las partes de una clase en Java.

Variables de clase

Las variables de clase se declaran de forma casi idéntica a las variables locales:

```
class AlgunaClase {
    int a;
}
```

Pueden utilizarse todos los tipos de Java (referencias incluidas), y son variables que se comportan exactamente igual que sus equivalentes locales. Lo que distingue a una variable de clase de una variable local (técnicamente), es su alcance y su *vida*.

El alcance de una variable de clase es *toda* la clase. La variable puede ser vista en casi todos los métodos de la clase (en un momento veremos en cuáles no), y utilizada o modificada dentro de todos ellos.

La vida de una variable es el tiempo en que la variable existe. En el caso de las variables locales, viven (o existen) desde el momento en que son declaradas, hasta que el método donde estén termine. Las variables de clase viven desde que el objeto es creado con new hasta que el objeto deja de existir.

Esto es importante; una variable de clase se crea al mismo tiempo que el objeto al que pertenece, y sigue siendo la misma variable aunque sean llamados varios métodos por el objeto. En cambio, una variable local existe dentro del método *y es distinta para cada llamada del método*. Cada vez que sea llamado un método, todas las variables locales declaradas dentro de él son creadas de nuevo. No son las mismas a través de distintas llamadas al método.

Aunque técnicamente hay pocas diferencias entre una variable local y una variable de clase, conceptualmente la diferencia es enorme en Orientación a Objetos. Las variables locales sólo sirven para hacer cuentas, u operaciones en general. Las variables de clase en cambio son fundamentales; son las que guardan el estado de un objeto, las que mapean el componente del mundo real al objeto con el que queremos representarlo en nuestro programa.

Vamos a añadirle variables de clase a nuestra clase Matriz2x2. Asumamos que los componentes de nuestra matriz están dispuestos de la siguiente manera:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

Entonces nuestra clase necesita cuatro valores en punto flotante para ser representada. Utilicemos el tipo **double**, para tener más precisión en nuestras operaciones. Y sólo para hacerlo más interesante, añadamos otro doble para que represente el determinante de nuestra matriz.

```
1  class Matriz2x2 {
2      double a;
3      double b;
4      double c;
5      double d;
6      double determinante;
7  }
```

Podríamos declararlas todas en una sola línea con **double a,b,c,d,determinante;** pero acostumbraremos declarar así las variables de clase, una por renglón.

Las variables de clase son únicas para cada objeto. Esto quiere decir que si tenemos dos objetos de nuestra clase Matriz2x2, digamos m1 y m2, entonces las variables a, b, c, d y determinante del objeto m1 *pueden tener valores totalmente distintos* a las variables correspondientes del objeto m2. Los estados de los objetos (en otras palabras, los valores de sus variables de clase) son independientes entre ellos. Si empezamos a modificar los valores de las variables del objeto m1, esto no afectará a las variables del objeto m2.

Métodos

En los métodos no pasa como con las variables, que hay locales y de clase. *Todas* las funciones (o métodos, o procedimientos) son de clase. No hay métodos locales.

Para entender cómo se declaran métodos en Java, declaremos en nuestra clase Matriz2x2 un método para calcular la multiplicación de una matriz por un valor escalar, pero no lo vamos a implementar (no lo vamos a escribir, todavía), sólo lo vamos a declarar. Como ya llevamos Álgebra Superior I, sabemos que

$$x\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} xa & xb \\ xc & xd \end{pmatrix}.$$

Un método tiene un nombre, un tipo de regreso, y parámetros. En Java se declaran así los métodos:

```
<tipoDeRegreso> <nombreDelMétodo> ( <parámetro1>,
                                    <parámetro2>,
```

```
                                                    ...
                                                    <parámetroN>) {
        <cuerpoDelMétodo>
    }
```

Cada parámetro es de la forma <tipo> <nombre>, donde el tipo es el tipo del parámetro, y el nombre es el nombre que tendrá dentro de la función.

A veces nuestros métodos no van a hacer ningún cálculo, sino que van a realizar acciones; en esos casos se pone que el tipo de regreso del método es **void**. Las expresiones que consisten de un objeto llamando a un método cuyo tipo de regreso es **void** no tienen valor. Son expresiones que no pueden usarse del lado derecho de una asignación, o para pasarlas como parámetros a una función.

El cuerpo del método para multiplicar matrices con escalares no lo haremos de inmediato, sino un poco más adelante.

Como carecemos de originalidad, vamos a llamar a nuestro método multiplica, ya que va a multiplicar a nuestra matriz con un valor escalar. Regresa una matriz, ya que la multiplicación de una matriz con un escalar regresa una matriz. Y recibe un escalar (en este caso un **double**), con el que multiplicaremos a nuestra matriz. Así que la declaración queda así

```
5       Matriz2x2 multiplica (double x) {
6       }
```

Puedes intentar compilar la clase, pero no va a funcionar porque el método multiplica debe regresar una referencia de tipo Matriz2x2, y no regresa nada (de hecho no hace nada). Si quieres que la clase compile, puedes hacer que regrese **null** por ahora, y cuando escribamos el método ya haremos que regrese lo que tiene que regresar:

```
5       Matriz2x2 multiplica (double x) {
6         return null;
7       }
```

Varios de ustedes deben haber observado algo raro. Vamos a multiplicar una matriz y un escalar; entonces ¿por qué sólo le pasamos un escalar a la función, y no le pasamos una matriz también?

La respuesta es que dentro de la función multiplica, ya hay una matriz; la que manda llamar el método.

Cuando queramos *usar* a la función multiplica, necesitaremos un objeto de la clase Matriz2x2 para que la llame. Supongamos que ya existe este objeto y que se llama m. Para llamar al método, tendremos que hacer:

```
m.multiplica (2.5);
```

Entonces, dentro del método tendremos el **double** x (con valor 2.5), y a m, que es pasada implícitamente porque es el objeto que mandó llamar a la función. Dentro del método, m se va a llamar **this**; *éste*, en inglés, lo que hace énfasis en que *éste* mandó llamar la función.

La fuerte tipificación de Java que vimos la práctica pasada se aplica también a los métodos. El tipo de un método está determinado por el tipo del valor que regresa, y el tipo de cada uno de sus parámetros.

Por lo tanto, lo siguiente es ilegal:

```
m.multiplica (true);
```

Sin embargo, todo esto sí es legal:

```
double y = 3.7;
float  x = 23.14;
short  s = 5;

m.multiplica (y);
m.multiplica (x);
m.multiplica (s);
m.multiplica (1.3);
m.multiplica (4);
```

por lo que dijimos la práctica anterior de que hay tipos que sí caben dentro de otros tipos. Lo siguiente en cambio es ilegal:

```
char c = m.multiplica (2.9);
```

El método multiplica regresa un valor de tipo referencia a un objeto de la clase Matriz2x2. No podemos asignárselo a una variable de tipo char. Esto sí es correcto:

```
Matriz m2;
double w = 3.8;
m2 = m.multiplica (w);
```

y de hecho así será usado casi siempre. En la variable m2 quedará guardada la matriz resultante de multiplicar m con 3.8 (cuando escribamos el cuerpo del método; como está ahora, m2 terminaría valiendo null).

En los ejemplos vimos que le podemos pasar tanto literales como variables al método. Esto es importante; las funciones en Java reciben sus parámetros *por valor*. Esto quiere decir que cuando la función es llamada, no importa qué haya dentro de los paréntesis (una variable, una literal, cualquier expresión), el valor de lo que haya *se copia*, y se le asigna al parámetro dentro de la función (en nuestro caso, x). Por supuesto, ese valor debe ser de un tipo compatible al del parámetro.

Dentro de la función, x se comporta de manera idéntica a una variable local; pero es inicializada cuando se manda llamar el método, con el valor que le pasamos. Lo mismo pasa con todos los parámetros de cualquier función.

Supongamos que mandamos llamar a multiplica así:

```
double x = 14.75;
m.multiplica (x);
```

y supongamos nuestro método multiplica fuera así

```
5       Matriz2x2 multiplica (double x) {
6         x = x / 5;
```

```
7           return null;
8       }
```

La variable x que le pasamos al método cuando lo llamamos, es totalmente independiente del parámetro x del método. La variable x no se ve afectada cuando le hacemos x = x / 5 al parámetro x dentro del método. La variable x seguirá valiendo 14.75;

Escribamos el cuerpo del método multiplica

```
5       Matriz2x2 multiplica (double x) {
6           double a, b, c, d;
7
8           a = x * this.a;
9           b = x * this.b;
10          c = x * this.c;
11          d = x * this.d;
12      }
```

Noten que declaramos cuatro variables locales que tienen el mismo tipo y se llaman igual que variables de clase existentes. No importa, porque usamos **this** para distinguir las variables de clase de las locales.

Con esto, ya tenemos los cuatro componentes necesarios (a, b, c, d) para que hagamos una nueva matriz. El único problema es que no sabemos hacer matrices.

Para hacer una matriz, vamos a necesitar un constructor.

Constructores

Conceptualmente, los constructores sirven para que definamos el estado cero de nuestros objetos; el estado que tendrán al ser creados. Para motivos prácticos, los constructores son sencillamente métodos que no tienen tipo de regreso, y que sólo podemos llamar a través del operador **new**.

Como los métodos, pueden recibir parámetros (por valor también), y puede hacerse dentro de ellos cualquier cosa que podamos hacer dentro de un método, excepto *regresar* un valor.

Un constructor obvio para nuestras matrices sería así:

```
5       Matriz2x2 (double a, double b, double c, double d) {
6           this.a = a;
7           this.b = b;
8           this.c = c;
9           this.d = d;
10
11          this.determinante = this.a * this.d - this.b * this.c;
12      }
```

(En tu archivo `Matriz2x2.java`, y en todas las clases que escribas, pon siempre los constructores antes que los métodos).

Observen que el constructor recibe parámetros que se llaman igual y que tienen el mismo tipo de las variables de clase. No importa, porque usamos this para tener acceso a las variables

de clase, y así ya no hay confusión. En los constructores, **this** hace referencia al objeto que está siendo construido.[2]

En este caso el constructor inicializa las variables de clase con los valores que recibe, y calcula el determinante de la matriz (qué bueno que recordamos que $\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$).

Ahora ya podemos terminar nuestro método multiplica:

```
15      Matriz2x2 multiplica (double x) {
16          double a, b, c, d;
17
18          a = x * this.a;
19          b = x * this.b;
20          c = x * this.c;
21          d = x * this.d;
22
23          Matriz2x2 resultado;
24          resultado = new Matriz2x2 (a, b, c, d);
25          return resultado;
26      }
```

Nuestro método multiplica ya está completo.

Polimorfismo

Creemos un nuevo método para multiplicar matrices. Como son de dos por dos, se pueden multiplicar y el resultado es una matriz de dos por dos:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} ax + bz & cx + dz \\ ay + bw & cy + dw \end{pmatrix}.$$

El método va a recibir una matriz como parámetro (la otra matriz será la que llame la función), y va a regresar otra matriz.

Ahora, el orden importa en la multiplicación de matrices (no es conmutativa), y alguna de las matrices que vayamos a multiplicar tendrá que llamar al método, así que debemos definir cuál será; si el operando derecho o el izquierdo.

Como va a ser usado así el método:

```
Matriz2x2 m1, m2;
m1 = new Matriz2x2 (1,2,3,4);
m2 = new Matriz2x2 (5,6,7,8);

Matriz2x2 mult;
mult = m1.multiplica (m2); // Multiplicamos m1 por m2.
```

vamos a definir que la matriz que llame la función sea el operando izquierdo.

[2]Con nuestra clase Matriz2x2 hemos estado usando mucho **this** para no confundirnos con los nombres de las variables. Pero cuando no haya conflicto entre variables de clase y variables locales, podemos usar los nombres de las variables de clase sin el **this** y el compilador de Java sabrá que hacemos referencia a las variables de clase.

Aquí algunos estarán exclamando "¡hey, ya teníamos un método que se llama multiplica!". Es verdad; pero en Java podemos repetir el nombre de un método, siempre y cuando el número o tipo de los parámetros no sea el mismo (si no, vean en la documentación generada de la clase Consola y busquen el método imprime... mejor dicho los métodos imprime). A esta característica se le llama *polimorfismo*.

Nuestro (segundo) método multiplica quedaría así:

```
30      Matriz2x2 multiplica (Matriz2x2 m) {
31          double a, b, c, d;
32
33          a = this.a * m.a + this.b * m.c;
34          b = this.a * m.b + this.b * m.d;
35          c = this.c * m.a + this.d * m.c;
36          d = this.c * m.b + this.d * m.d;
37
38          Matriz2x2 resultado;
39          resultado = new Matriz2x2 (a, b, c, d);
40          return resultado;
41      }
```

Noten que usamos al operador . (punto) para tener acceso a las variables de clase del objeto m, que es el parámetro que recibe el método.

Una nota respecto al parámetro m. Como todos los parámetros, es una *copia* del valor que le pasaron. Pero los valores de las referencias son direcciones de memoria. Así que aunque esta dirección de memoria sea una copia de la que le pasaron, la dirección en sí es la misma. Hace referencia al mismo objeto (a la misma matriz) que se le pasó al método; por lo tanto, si modificamos la matriz dentro de la función, *sí se afecta* al objeto con el que se llamó la función. Hay que tener eso en cuenta.

El polimorfismo también se aplica a los constructores. Podemos tener tantos constructores como queramos, siempre que reciban un número de parámetros distintos o que sus tipos sean diferentes. Por ejemplo, podemos tener un constructor para nuestra matriz cero:

```
15      Matriz2x2 () {
16          this.a = 0;
17          this.b = 0;
18          this.c = 0;
19          this.d = 0;
20          this.determinante = 0;
21      }
```

Si sólo vamos a tener un constructor, y éste sólo va a poner en cero las variables (o en null, o en false), podemos no hacerlo. Java proporciona un constructor por omisión, que no recibe parámetros, y que inicializa todas las variables de clase como cero, si son numéricas, como false, si son booleanas, como '\u0000' si son caracteres, y como null si son referencias. Sin embargo, este constructor sólo está disponible si no hay ningún otro constructor en la clase.

Además, en estas prácticas siempre haremos constructores para nuestras clases, aunque hagan lo mismo que el constructor por omisión, es una buena práctica de programación.

Con todos estos conocimientos, hacer el resto de las operaciones para matrices de dos por

dos es trivial.

Actividad 4.2 Haz los métodos suma, resta e inversa para tu clase Matriz2x2. Las primeras dos son triviales (puedes basarte en el método para multiplicar matrices).
Para obtener la matriz inversa, primero comprueba que exista, o sea que el determinante sea distinto de cero (utiliza un if), si no, una división por cero hará que tu programa no funcione.

Clases de uso

Para probar nuestra clase Matriz2x2 podríamos sencillamente colocarle un método main como los que hemos usado. Empero, aunque esto es útil para probar inicialmente una clase, siempre hay que probarla desde otra clase.

La razón es que siempre que hagamos una clase, debemos hacerla pensando que es un componente más de algún problema (y si tenemos suerte, de varios problemas). Para que un componente sea realmente útil, debe poder incluirse en cualquier proyecto sin que haya que hacerle modificaciones, y si sólo probamos una clase dentro de ella misma, lo más seguro es que esta modularidad no sea bien comprobada.

Para probar nuestras clases, y para hacer nuestros programas, siempre utilizaremos *clases de uso*. Las clases de uso nunca tendrán variables de clase ni constructores. Muchas veces incluso consistirán sólo de una función main, como la clase UsoReloj de las primeras dos prácticas (para distinguirlas de nuestras clases "normales", a nuestras clases de uso les vamos a añadir el prefijo Uso).

Nuestra clase UsoMatriz2x2 quedaría así:

```
1   class UsoMatriz2x2 {
2       public static void main (String[] args) {
3           Matriz2x2 m1, m2;
4           m1 = new Matriz2x2 (1,2,3,4);
5           m3 = new Matriz2x2 (5,6,7,8);
6
7           Matriz mult;
8           mult = m1.multiplica (m2);
9       }
10  }
```

Pueden compilarla y ejecutarla. Parecerá que no hace nada, pero sí hace; lo que pasa es que no hemos creado una interfaz para mostrar nuestras matrices en la consola (y de hecho, ni siquiera declaramos una consola en nuestra clase de uso).

Para mostrar nuestras matrices en la consola, podríamos crear el método imprime en la clase Matriz2x2:

```
50  void imprime (Consola c) {
51      c.imprimeln (this.a);
52      c.imprimeln (this.b);
```

```
53          c.imprimeln (this.c);
54          c.imprimeln (this.d);
55      }
```

Tendríamos por supuesto que añadir **import** icc1_1.interfaz.Consola; a la clase Matriz2x2 para que compilara. Sin embargo, esto es un error.

Si metemos el método imprime en nuestra clase Matriz2x2, entonces ya no sería una clase independiente; dependería de la clase Consola. A cualquier lugar que lleváramos nuestra clase Matriz2x2, tendríamos que llevar también a la clase Consola.

A veces no podrá evitarse eso; una clase dependerá de otras muchas. Pero en este caso en particular, podemos hacer independiente a la clase Matriz2x2 de la clase Consola. Para ello, en lugar de pasarle *toda* una consola a la clase Matriz2x2 para que se imprima, mejor hagamos que la clase Matriz2x2 pueda ser representada en una cadena para que la imprimamos con la clase Consola. Esto lo logramos implementando el método toString:

```
50      public String toString () {
51          return this.a + "\t" + this.b + "\n" +
52              this.c + "\t" + this.d;
53      }
```

El **public** es obligatorio. Por qué debe llamarse exactamente toString lo explicaremos cuando veamos herencia, y en un momento veremos por qué estamos sumando dobles con cadenas. Por ahora, con este método en la clase, podemos imprimir nuestras matrices haciendo sencillamente:

```
Consola c;
c = new Consola ("Matrices");

Matriz2x2 m;
m = new Matriz2x2 (1,2,3,4);

c.imprimeln (m); // Podemos pasar a la matriz directamente.
```

Nuestra clase de uso quedaría ahora así:

```
1   import icc1_1.interfaz.Consola;
2
3   class UsoMatriz2x2 {
4       public static void main (String[] args) {
5           Consola c;
6           c = new Consola ("Matrices");
7
8           Matriz2x2 m1, m2;
9           m1 = new Matriz2x2 (1,2,3,4);
10          m2 = new Matriz2x2 (5,6,7,8);
11
12          c.imprimeln (m1);
13          c.imprimeln (m2);
14
15          Matriz mult;
```

```
16              mult = m1.multiplica (m2);
17
18              c.imprimeln (mult);
19          }
20      }
```

Compilen y ejecuten (recuerden que tienen que usar la opción `-classpath`, y que tienen que poner el archivo `interfaz1.jar` en el directorio donde estén trabajando).

Modificadores de acceso

¿Qué pasa si hacemos lo siguiente en el método main de nuestra clase de uso?

```
4           public static void main (String[] args) {
5               Consola c;
6               c = new Consola ("Matrices");
7
8               Matriz2x2 m;
9               m = new Matriz2x2 (1,2,3,4);
10
11              m.a = 10;
12              c.imprimeln (m.determinante);
13          }
14      }
```

El valor que se imprima en la consola será -2, aunque el determinante de nuestra matriz (después de la modificación que le hicimos en la línea 11) sea 34, porque el determinante es calculado cuando el objeto es creado.

Lo que pasa aquí es que el estado de nuestro objeto queda inconsistente. Y eso pasa porque modificamos directamente una variable de clase de nuestro objeto m. Podríamos hacer la solemne promesa de nunca modificar directamente una variable de clase; pero eso no basta. Tenemos que *garantizar* que eso no ocurra. No debe ocurrir *nunca*.

A esto se le llama el *acceso* de una variable de clase (que no es lo mismo que el alcance). Una variable puede tener acceso *público*, de *paquete*, *protegido* y *privado*. Las variables locales no tienen modificadores de acceso ya que sólo existen dentro de un método.

Cuando una variable de clase tiene acceso público, en cualquier método de cualquier clase puede ser vista y modificada. Cuando tiene acceso privado, sólo dentro de los métodos y constructores de la clase a la que pertenece puede ser vista y modificada. Los accesos de paquete y protegidos los veremos cuando veamos paquetes y herencia, respectivamente.

Restringir el acceso a una variable de clase nos permite controlar cómo y cuándo será modificado el estado de un objeto. Con esto podemos garantizamos la consistencia del estado, pero también nos permite cambiar las variables privadas de un objeto, y que éste siga siendo válido para otras clases que lo usen; como nunca ven a las variables privadas, entonces no importa que las quitemos, les cambiemos el nombre o tipo o le añadamos alguna. El acceso es el principal factor del encapsulamiento de datos.

Por todo esto, siempre es bueno tener las variables de clase con un acceso privado. Vamos a cambiarle el acceso a las variables de clase de Matriz2x2. Nuestras variables no tenían ningún

acceso especificado, por lo que Java les asigna por omisión el acceso de paquete; para motivos prácticos y con lo que hemos visto, el acceso es público.

Para cambiárselo a privado, sólo tenemos que hacer

```
3  class Matriz2x2 {
4      private double a;
5      private double b;
6      private double c;
7      private double d;
8      private double determinante;
```

Los modificadores para público, protegido y de paquete son **public**, **protected** y **package** respectivamente. Aunque no es necesario poner **package** para que una variable de clase tenga acceso de paquete, en estas prácticas será requisito que el acceso de los métodos y las variables sea explícito.

Los métodos también tienen acceso, y también es de paquete por omisión. El acceso en los métodos tiene el mismo significado que en las variables de clase; si un método es privado, no podemos usarlo más que dentro de los métodos de la clase, y si un método es público, se puede usar en cualquier método de cualquier clase.

Los métodos privados son generalmente funciones auxiliares de los métodos públicos y de paquete, y al hacerlos privados podemos agregar más o quitarlos cuando queramos, y las clases que usen a nuestra clase no se verán afectadas. El encapsulamiento de datos facilita la modularización de código.

Vamos a cambiar el acceso de todos los métodos de nuestra clase Matriz2x2 a público, porque eso es lo que queremos, que toda matriz pueda utilizar sus métodos no importa dónde esté declarada. Para cambiar el acceso, sólo agrega el modificador **public** antes de cada declaración, como por ejemplo:

```
30     public Matriz2x2 multiplica (Matriz2x2 m) {
31         ...
```

También será requisito que todos los métodos tengan el acceso explícito, aunque sea de paquete.

Los constructores también tienen acceso, y como en muchas otras cosas funcionan de manera idéntica que para métodos. Sin embargo, si tenemos un constructor privado, significa que no podemos construir objetos de esa clase fuera de la misma clase; no podríamos construir objetos de la clase en una clase de uso, por ejemplo. Veremos en un momento por qué querríamos hacer eso.

Hay que cambiar también el acceso de nuestros constructores a público, porque por omisión también es de paquete.

```
5      public Matriz2x2 (double a, double b, double c, double d) {
6          ...
```

No sólo las variables de clase, los métodos y constructores tienen acceso. Las mismas clases tienen acceso, y también por omisión es de paquete. Una clase privada parece no tener sentido; y no obstante lo tiene. Veremos más adelante cuándo podemos hacer privado el acceso a una

clase; por ahora, cambiemos el acceso de nuestra clase a público:

```
3   public class Matriz2x2 {
4           ...
```

Ya que nuestras variables de clase son privadas, ¿cómo hacemos para poder verlas o modificarlas fuera de la clase?

Verlas es lo más sencillo; creamos un método para cada variable de clase privada que nos regrese su valor:

```
70      public double dameA () {
71          return this.a;
72      }
73
74      public double dameB () {
75          return this.b;
76      }
77
78      ...
```

Y para modificarlas, hacemos un método para cada una de ellas también:

```
90      public void defineA (double a) {
91          this.a = a;
92          this.determinante = this.a * this.d - this.b * this.c;
93      }
```

Puede parecer engorroso; pero la justificación se hace evidente en esta última función. Cada vez que modifiquemos un componente de la matriz, actualizaremos automáticamente el valor del determinante. Con esto dejamos bien definido cómo ver y cómo modificar el estado de un objeto, y garantizamos que sea consistente siempre.

El acceso es parte fundamental del diseño de una clase. Si hacemos un método o variable de clase público, habrá que cargar con él siempre, ya que si lo quitamos o lo hacemos privado, es posible que existan aplicaciones que lo utilizaran, y entonces éstas ya no servirían.

Actividad 4.3 Implementa los métodos dameA, dameB, dameC, dameD y dameDeterminante y defineA, defineB, defineC y defineD.
Puedes observar que no habrá método defineDeterminante, ya que el determinante depende de las componentes de la matriz.

Otros modificadores

Hay dos modificadores más para las variables y métodos de Java. El primero de ellos es **final**. Una variable final sólo puede ser inicializada una vez; y generalmente será al momento de ser declarada. Una vez inicializada, ya no cambia de valor nunca. Su valor es *final*. Las variables finales pueden ser de clase o bien locales.

Cuando tengamos una literal que utilizamos en muchas partes del código, y que dentro de la ejecución del programa nunca cambia (por ejemplo, el IVA en un programa de contabilidad), siempre es mejor declarar una variable final para definirlo, y así si por alguna razón nos vuelven a subir el IVA,[3]en lugar de cambiar *todos* los lugares donde aparezca la literal, sólo cambiamos el valor que le asignamos a la variable final. También facilita la lectura del código.

Otros lenguajes de programación tienen *constantes*. Las variables finales de Java no son constantes exactamente; pero pueden emular a las constantes de otros lenguajes. Hay métodos finales también, pero su significado lo veremos cuando veamos herencia.

El otro modificador es **static**. Una variable de clase estática es la misma para *todos* los objetos de una clase. No es como las variables de clase normales, que cada objeto de la clase tiene su propia copia; las variables estáticas son compartidas por todos los objetos de la clase, y si un objeto la modifica, todos los demás objetos son afectados también. No hay variables locales estáticas (no tendrían sentido).

También hay métodos estáticos. Un método estático no puede tener acceso a las variables de clase no estáticas; no puede hacer **this**. a ninguna variable. El método main por ejemplo siempre es estático.

Para tener acceso a una variable estática o a un método estático se puede utilizar cualquier objeto de la clase, o la clase misma. Si la clase Matriz2x2 tuviera una variable estática var por ejemplo, podríamos hacer Matriz2x2.var para tener acceso a ella (si var fuera pública, por supuesto).

Hay clases cuyos objetos necesitan pasar por un proceso de construcción más estricto que las clases normales. Para este tipo de clases, que suelen llamarse *clases fábrica* (o *factory classes* en inglés), lo que se hace es que tienen constructores privados, y entonces se usan métodos estáticos para que creen los objetos de la clase usando las restricciones que sean necesarias.

Si queremos una variable privada, que además sea estática, y además final, hay que hacer

```
public static final int n = 0; // ¡Hay que inicializar si es final!
```

por ejemplo. No se puede cambiar el orden; lo siguiente no compila

```
public final static int n = 0; // ¡Hay que inicializar si es final!
```

Básicamente eso es todo lo necesario para escribir una clase.

4.3.2. Cadenas

La práctica anterior vimos tipos básicos, pero varios de ustedes han de haber observado que constantemente mencionábamos "cadenas" sin nunca definirlas. Por ejemplo, al constructor de la clase Consola le pasamos una cadena:[4]

```
Consola c;
c = new Consola ("Título␣de␣mi␣consola");
```

Las cadenas (*strings* en inglés) son objetos de Java; pero son los únicos objetos que tienen literales distintas de **null** y definido el operador +. Además, son los únicos objetos que podemos

[3]O más raro; que lo bajen.

[4]Observa que estamos utilizando ␣ para representar los espacios dentro de las cadenas.

crear sin utilizar new. Las cadenas son consideradas por todo esto un tipo básico; pero no por ello dejan de ser objetos, con todos los atributos y propiedades de los objetos.

Las literales de las cadenas las hemos usado a lo largo de estas prácticas; son todos los caracteres que están entre comillas:

```
"hola mundo";
"adiós..."
"En un lugar de la Mancha de cuyo nombre..."
```

Para declarar una variable de tipo cadena, declaramos un objeto de la clase String:

```
String a;
```

pero no es necesario instanciarlo con new; el asignarle una literal de cadena basta. Las dos instanciaciones de abajo son equivalentes

```
a = "hola mundo";
a = new String ("hola mundo");
```

Esta es una comodidad que Java otorga; pero en el fondo las dos instanciaciones son exactamente lo mismo (en ambas la JVM asigna memoria para el objeto), aunque en la primera no tengamos que escribir el new String ( ).

Además de tener literales e instanciación implícita, los cadenas son el único objeto en Java que tiene definido el operador +, y sirve para concatenar cadenas:

```
String h, m, a;

h = "hola ";
m = "mundo";
a = h + m;
```

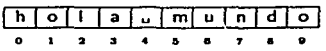
En este ejemplo, la variable a termina valiendo "hola mundo". Lo bonito del caso es que no sólo podemos concatenar cadenas; podemos concatenar cadenas con cualquier tipo de Java. Si hacemos

```
int edad = 15;
String a;

a = "Edad: " + 15;
```

la cadena resultante es "Edad: 15". Y funciona para enteros, flotantes, booleanos caracteres e incluso referencias. Por eso nuestro método toString podía "sumar" dobles con cadenas; realmente estaba concatenando cadenas.

Por dentro, una cadena es una secuencia de caracteres uno detrás del otro. La cadena a que vimos arriba tiene esta forma por dentro:



Las cadenas son objetos de sólo lectura. Esto quiere decir que podemos tener acceso al carácter 'm' de la cadena "hola␣mundo", pero no podemos cambiar la m por t para que la cadena sea "hola␣tundo".

Sin embargo, podemos crear un *nuevo* objeto cadena que diga "hola␣tundo" a partir de "hola␣mundo", y asignarlo de nuevo a la variable:

```
String a;
a = "hola␣mundo";
a = a.substring (0,5) + "t" + a.substring (6,10);
```

Ahora a representa a "hola␣tundo"; pero hay que entender que a a se le asignó un nuevo objeto (construido a partir del original), no que se modificó el anterior (y además, utiliza tres objetos temporales para construir el nuevo objeto; el primero a.substring (0,5), el segundo "t", y el tercero a.substring (6,10)).

Las cadenas son parte fundamental de la aritmética de Java; ofrecen muchas funciones y utilidades que en muchos otros lenguajes de programación es necesario implementar a pie.

Regresando a la cadena "hola␣tundo", su longitud es 10 (tiene 10 caracteres), y siempre puede saberse la longitud de una cadena haciendo

```
int longitud = a.length ();
```

Las cadenas pueden ser tan grandes como la memoria lo permita.

Cada uno de los caracteres de la cadena pueden ser recuperados (leídos), pero no modificados. Java ofrece además muchas funciones para construir cadenas a partir de otras cadenas.

Por ejemplo, si queremos sólo la subcadena "tundo", sólo necesitamos hacer

```
String b;
b = a.substring (5,10);
```

De nuevo; esto regresa un nuevo objeto sin modificar el anterior. La cadena a seguirá siendo "hola␣tundo".

Hay que notar que substring regresa todos los caracteres entre el índice izquierdo y el índice derecho *menos uno*. Esto es para que la longitud de la subcadena sea el índice derecho menos el índice izquierdo.

Puede que sólo deseemos un carácter, entonces podemos usar

```
char c = a.charAt (5);
```

Y esto regresa el carácter (tipo char) 'm'. Si tratamos de leer un carácter en una posición inválida (menor que cero o mayor o igual que la longitud de la cadena), el programa terminará con un error.

Las variables de la clase String son referencias, como todos los objetos. Eso quiere decir que cuando hagamos

```
String a = "hola␣mundo";
String b = "hola␣mundo";

boolean c = (a == b);
```

el **boolean** c tendrá como valor **false**. Las variables a y b son *distintas*; apuntan a direcciones distintas en la memoria. Que en esas direcciones haya cadenas idénticas no le importa a Java; lo importante es que son dos referencias *distintas*. Si queremos comparar dos cadenas *lexicográficamente*, debemos usar el método equals de la clase String:

```
c = a.equals (b);
```

En este caso, el valor de c sí es **true**.

Las literales de cadena en Java son objetos totalmente calificados. Por lo tanto, es posible llamar métodos utilizándolas (por bizarro que se vea):

```
int l = "hola␣mundo".length ();
String b;
b = "hola␣mundo".substring (0,4);
```

La clase String de Java ofrece muchas otras funciones útiles, que iremos aprendiendo a usar a lo largo de estas prácticas.

Actividad 4.4 Con la ayuda del ayudante, consulta la documentación de la clase String. Necesitarás estarla consultando para los ejercicios de esta práctica.

4.3.3. El recolector de basura

Hemos dicho que cuando utilizamos el operador **new** para instanciar un objeto, la JVM le asigna una porción de la memoria para que el objeto sea almacenado. También dijimos que las variables de clase viven desde que el objeto es declarado, hasta que éste deja de existir.

Y en el ejemplo de "hola␣mundo" dijimos que al hacer

```
String a;
a = "hola␣mundo";
a = a.substring (0,5) + "t" + a.substring (6,10);
```

estábamos usando tres objetos temporales (las tres subcadenas que forman "hola␣tundo").

¿Qué pasa con esas subcadenas? ¿Cuándo deja de existir un objeto? Si **new** hace que la JVM asigne memoria, ¿cuándo es liberada esta memoria?

La respuesta corta es *no hay que preocuparse por ello*. La larga es así: imagínense que cada objeto tiene un contador interno que se llama *contador de referencias*. Cuando un objeto es instanciado con new (aunque sea implícitamente, como las cadenas), ese contador se pone en cero. En cuanto una referencia comienza a apuntar hacia el objeto, el contador aumenta en uno. Por cada referencia distinta que empiece a apuntar hacia el objeto, el contador se aumentará en uno. Por ejemplo

```
String a;
a = "cadena";

String b, c, d, e;

b = c = d = e = a;
```

En este ejemplo, al objeto de la clase cadena que instanciamos como "cadena", su contador de referencias vale 5, ya que a, b, c, d y e, apuntan hacia él.

Cada vez que una referencia deje de apuntar al objeto, el contador disminuye en 1. Si ahora hacemos

```
a = b = c = d = e = null;
```

con eso las cinco referencias dejan de apuntar al objeto, y entonces su contador de referencias llega a cero.

Cuando el contador de referencias llega a cero, el objeto es marcado como "removible". Un programa especial que corre dentro de la JVM (el recolector de basura) se encarga cada cierto tiempo de buscar objetos marcados como removibles, y regresar al sistema la memoria que utilizaban.

Debe quedar claro que este es un ejemplo simplificado de cómo ocurre realmente la recolección de basura; en el fondo es un proceso mucho más complejo y el algoritmo para marcar un objeto como removible es mucho más inteligente. Pero la idea central es esa; si un objeto ya no es utilizado, que el recolector se lo lleve.

La respuesta corta es suficiente sin embargo. La idea del recolector de basura es que los programadores (ustedes) no se preocupen de liberar la memoria que asignan a sus objetos. En otros lenguajes eso tiene mucha importancia; mas por suerte Java lo hace por nosotros.

4.3.4. Comentarios para JavaDoc

Nuestra clase Matriz2x2 es una clase terminada y funcional. Otros programadores podrían querer utilizarla para sus propios programas. Para ayudarlos a entender como funciona la clase, podemos hacer disponibles para ellos la documentación generada por JavaDoc.

Actividad 4.5 En el directorio donde esté tu clase Matriz2x2, crea un directorio llamado `doc/` y corre `javadoc` de la siguiente manera:

```
# mkdir doc/
# javadoc -d doc/ Matriz2x2.java
```

Con la ayuda de tu ayudante, revisa la documentación que se genera.

Puedes ver que la documentación generada es bastante explícita por sí misma. Nos dice el tipo de regreso de los métodos, así como sus parámetros, y qué constructores tiene la clase.

Con esa información cualquiera que sepa lo que nosotros sabemos puede empezar a trabajar con una clase.

Sin embargo, podemos hacer todavía más para que la documentación sea aún más explícita. Podemos utilizar comentarios especiales para que JavaDoc haga la documentación más profunda.

Para que JavaDoc reconozca que un comentario va dirigido hacia él, necesitamos comenzarlos con /** en lugar de sólo /*. Después de eso, sólo necesitamos poner el comentario inmediatamente antes de la declaración de un método, un constructor o de una clase para que JavaDoc lo reconozca. Modifica así la declaración de la clase Matriz2x2

```
3  /**
4   * Clase para representar matrices de dos renglones por dos columnas.
5   */
6  public class Matriz2x2 {
7      ...
```

y modifica así la declaración del método multiplica (el que multiplica por escalares):

```
15     /**
16      * Calcula el resultado de multiplicar la matriz por un escalar.
17      */
18     public Matriz2x2 multiplica (double x) {
19         ...
```

Vuelve a generar la documentación de la clase y consúltala para que veas los resultados.

Para métodos, podemos mejorar todavía más la documentación utilizando unas etiquetas especiales. Vuelve a modificar el comentario del método multiplica, pero ahora así:

```
15     /**
16      * Calcula el resultado de multiplicar la matriz por un escalar.
17      * @param x el escalar por el que se multiplicará la matriz.
18      * @return la matriz resultante de multiplicar esta matriz por
19      *         el escalar x.
20      */
21     public Matriz2x2 multiplica (double x) {
22         ...
```

Vuelve a generar la documentación y comprueba los cambios. La etiqueta @param le sirve a JavaDoc para saber que vamos a describir uno de los parámetros del método. El nombre del parámetro debe corresponder a alguno de los parámetros que recibe el método, obviamente. Si no, JavaDoc marcará un error, aunque generará la documentación de todas maneras.

La etiqueta @return le dice a JavaDoc que se va a describir qué es lo que regresa el método.

Todos los métodos que escribas de ahora en adelante, deberán estar documentados para JavaDoc, con su valor de regreso (si tiene el método), y cada uno de los parámetros que reciba, si recibe. Las clases también tienen que estar documentadas.

4.4. Ejercicios

Entre las aplicaciones más utilizadas en el mundo de la computación están las Bases de Datos. A partir de esta práctica, iremos recolectando todo lo que aprendamos para construir poco a poco una Bases de Datos para una agenda.

Una Base de Datos tiene registros, que a su vez están compuestos de campos. Los campos de nuestra Base de Datos para una agenda serán nombre, dirección y teléfono.

Por ahora, empezaremos con una Base de Datos ya llena, lo que quiere decir que los datos ya están dados y que no necesitarás introducirlos tú mismo.

Aunque hay muchas formas de implementar Bases de Datos, una de las más comunes es utilizar una tabla; los renglones son los registros, y las columnas campos, como en

Nombre	Dirección	Teléfono
Juan Pérez García	Avenida Siempre Viva # 40	55554466
Arturo López Estrada	Calle de la abundancia # 12	55557733
Edgar Hernández Levi	Oriente 110 # 14	55512112
María García Sánchez	Avenida Insurgentes Sur # 512	56742391
Pedro Páramo Rulfo	Avenida México Lindo # 23	54471499
José Arcadio Buendía	Macondo # 30	56230190
Florentino Ariza	Calle de la Cólera # 11	55551221
Galio Bermúdez	Sótanos de México # 45	55552112
Carlos García Vigil	La República # 1	55554332
Eligio García Agusto	Ciudades Desiertas # 90	56344325

Para representar a todos los datos, utilizaremos una sola variable estática de tipo cadena, llamada tabla. Los registros (o renglones) son de tamaño fijo, así que si queremos el primer registro sólo necesitamos hacer

```
tabla.substring (0,TAM_REGISTRO);
```

Donde TAM_REGISTRO es otra variable, estática y final, que tiene el tamaño del registro. Si necesitamos el segundo registro, tenemos que hacer

```
tabla.substring (TAM_REGISTRO, TAM_REGISTRO+TAM_REGISTRO);
```

En general, si necesitamos el i-ésimo registro (donde el primer registro es el registro 0, como buenos computólogos), hacemos

```
tabla.substring (i*TAM_REGISTRO, i*TAM_REGISTRO+TAM_REGISTRO);
```

Para obtener los campos se necesita algo similar usando los tamaños de cada campo. Nota que estamos usando TAM_REGISTRO solo, sin hacer this.TAM_REGISTRO o BaseDeDatosAgenda.TAM_REGISTRO. Esto es porque estamos dentro de la clase BaseDeDatosAgenda; cuando usamos TAM_REGISTRO el compilador asume que hablamos de las variables de la clase donde estamos.

Por ahora, asumiremos que todos los campos son únicos; o sea que no se repiten nombres, ni direcciones ni teléfonos.

1. Por ahora, nuestra Base de Datos de agenda será sólo de lectura, ya que no insertaremos nuevos registros ni borraremos ninguno.

   Sin embargo sí leeremos datos, y de hecho podremos hacer búsquedas.

   Abre el archivo `BaseDeDatosAgenda.java` donde está el esqueleto de lo que hemos estado discutiendo. Por ahora sólo tiene dos funciones (vacías); dameRegistroPorNombre y dameRegistroPorTelefono. La primera recibe una cadena con el nombre a buscar, y la segunda recibe un entero con el teléfono. Ambas regresan una cadena que es el registro que caza la búsqueda, o null si el nombre o teléfono no están en la Base de Datos.

   Por ejemplo, si hacemos

```
BaseDeDatosAgenda bdda;
String busqueda;

bdda = new BaseDeDatosAgenda ();

busqueda = bdda.dameRegistroPorNombre ("Pedro␣Páramo␣Rulfo");
```

   Entonces la cadena busqueda debe valer

   "Pedro Páramo Rulfo␣␣␣Avenida␣México␣Lindo␣␣23␣␣␣␣␣␣54471499"

   Y lo mismo si hacemos

```
busqueda = bdda.dameRegistroPorTelefono (54471499);
```

   Utilizando las funciones de la clase String, implementa los métodos.

   Una pista para ayudar: vas a necesitar convertir enteros en cadenas. Piensa que necesitarás el valor de un entero como cadena. Y vas a necesitar encontrar dónde empieza una subcadena dentro de otra cadena. Lee *todos* los métodos de la clase cadena (en la documentación generada de la clase String), antes de utilizar alguno.[5]

2. Queremos utilizar nuestra Base De Datos, así que escribe una clase de uso llamada UsoBaseDeDatosAgenda, que sólo tenga la función main y utilice las funciones que has escrito.

   Dentro de main debes crear un objeto de la clase BaseDeDatosAgenda, realizar una búsqueda por nombre, imprimir el resultado, realizar una búsqueda por teléfono e imprimir el resultado. Utiliza la clase Consola.

3. Utiliza comentarios de JavaDoc en los métodos dameRegistroPorNombre y dameRegistroPorTelefono para documentar qué es lo que deben hacer.

[5] Sí; la documentación está en inglés. Váyanse acostumbrando.

4.5. Preguntas

1. ¿Qué otras funciones debe implementar una Base de Datos de agenda?
2. ¿Podríamos implementarlas con lo que sabemos hasta ahora?
3. ¿Qué crees que necesitaríamos?

Estructuras de control y listas

A hardware failure will cause system software to crash, and the customer engineer will blame the programmer.

– Murphy's Laws of Computer Programming #5

Semana:	Séptima semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Estructuras de control, listas.
Objetivo:	Usar estructuras de control y listas.

Esta práctica se realiza durante la séptima semana del curso. El profesor debe haber visto las estructuras de control (que no debe llevarle mucho tiempo), y de ser posible haber comenzado con listas. El tema de listas debería ser visto durante toda la semana en que se realice la práctica.

Las estructuras de control no deberían presentarles ningún problema. Si se han familiarizado ya con la sintaxis de Java, las estructuras de control deberían poder ser comprendidas fácilmente.

La razón de ver listas en la quinta práctica del curso es una de las innovaciones que queremos dar al primer curso de programación utilizando Orientación a Objetos. Las listas son una estructura muy poderosa, que facilita el entender y manejar varios conceptos que deben verse en el curso: ciclos, recursión; incluso herencia, como veremos en la siguiente práctica.

Java además, por su recolector de basura y herencia, nos permite ver y utilizar listas como se ven en los lenguajes funcionales (incluso tenemos car y cdr; pero por suerte los nombres que usamos sí tienen sentido).

No hemos visto herencia, así que no podemos usar una lista universal, por lo que comenzamos con listas de cadenas, que es suficiente para los puntos que queremos cubrir en esta práctica.

Práctica 5
Estructuras de control y listas

5.1. Meta

Que el alumno aprenda a utilizar condicionales, iteraciones y listas.

5.2. Objetivos

Al finalizar la práctica el alumno será capaz de:

- utilizar condicionales (**switch**, **if**),
- utilizar iteraciones (**while**, **do ... while**, **for**), y
- utilizar listas.

5.3. Desarrollo

Un método no es sólo una serie de instrucciones ejecutándose una por una. Un método tiene que tomar decisiones a partir de los parámetros que reciba. Hemos utilizado el **if** para hacer esto; pero no lo hemos definido formalmente.

Además de tomar decisiones, un método puede necesitar repetir una tarea un determinado número de veces; o mientras cierta condición se cumpla.

En esta práctica, aprenderemos cómo nuestros métodos pueden tomar decisiones (varias decisiones a la vez, de hecho), y cómo repetir tareas, utilizando condicionales e iteraciones (también conocidas como ciclos).

Con estos nuevos conocimientos podremos comenzar a utilizar una de las estructuras más poderosas que hay en computación (independientemente del lenguaje): las listas simplemente ligadas.

5.3.1. Condicionales

En la práctica anterior, cuando una de nuestras búsquedas fallaba, el método debía regresar **null**.

Muchas funciones (y muchas más que escribiremos) regresarán un valor especial cuando algo no funcione exactamente como esperábamos. Hay que saber cuándo son regresados estos valores especiales y cómo tratarlos.

Ése es un ejemplo particular. En general, un método debe ser capaz de tomar decisiones de acuerdo a los parámetros que reciba. Para tomar decisiones, Java ofrece dos condicionales: **switch** e **if**.

La condicional switch

La condicional **switch** nos sirve para tomar múltiples decisiones a partir de una expresión de tipo **int**, **short**, **byte** o **char**.

Supongamos que tenemos una clase Prueba con las siguientes variables de clase:

```
public static final int SOLTERO     = 1;
public static final int CASADO      = 2;
public static final int DIVORCIADO  = 3;
public static final int VIUDO       = 4;
public static final int OTRO        = 5;
```

Con esto podemos clasificar de manera legible los distintos estados civiles. Y podemos usar un **switch** para tomar decisiones (si asumimos que el método dameEstadoCivil de alguna manera obtiene el estado civil):

```
Consola c;
c = new Consola ();

Prueba p;
p = new Prueba ();

... // Aquí hacemos varias cosas con p.

int estadoCivil = p.dameEstadoCivil ();

switch (estadoCivil) {
  case Prueba.SOLTERO:
    c.imprimeln ("Este␣sí␣es␣listo.");
    break;
  case Prueba.CASADO:
    c.imprimeln ("A␣éste␣ya␣lo␣agarraron.");
    break;
  case Prueba.DIVORCIADO:
    c.imprimeln ("Este␣recapacitó.");
    break;
  case Prueba.VIUDO:
    c.imprimeln ("Este␣tomó␣decisiones␣extremas.");
    break;
  default:
    c.imprimeln ("Este␣no␣quiere␣decir.");
}
```

La condicional **switch** sólo recibe expresiones de alguno de los tipos que dijimos arriba; no funciona con expresiones de punto flotante, booleanas o siquiera de tipo **long**.

Una vez que recibe la expresión, calcula su valor y comprueba si algún **case** caza con ese valor. Si caza, se ejecuta el código a partir de los dos puntos, *y hasta el final del* **switch**. Esto es importante; se ejecuta desde el **case** que caza con el valor recibido, y se siguen ejecutando todos los **case** que le siguen. Para evitar que se sigan ejecutando todos los **case**, ponemos el **break**. El **break** (como su nombre indica) *rompe* la ejecución del bloque donde estemos.

Si sólo queremos que se ejecute uno de los **case**, siempre hay que poner **break**; pero a veces querremos que se ejecute más de un **case**.

Si ningún **case** caza con el valor recibido, se ejecuta el **default**. No es obligatorio que exista un **default**, pero se recomienda para que absolutamente todos los casos queden cubiertos. Si no hay **default**, sencillamente el programa continúa con la ejecución del enunciado que le siga al **switch**.

En este ejemplo, usamos variables finales para etiquetar los **case**. Es el único tipo de variables que podemos usar: un **case** sólo puede ser etiquetado por una variable final o una literal (de los tipos que mencionamos arriba).

La condicional **switch** nos permite tomar decisiones con varias opciones. Muchas veces, sin embargo, vamos a necesitar decidir sólo entre sí y no. Para eso está el **if**.

La condicional if ... else

Ya hemos utilizado la condicional **if**; nos permite tomar decisiones binarias: sí o no.

Por ejemplo, para comprobar que la función dameRegistroPorNombre no regresara **null** debíamos hacer algo así:

```
String reg;
reg = bdd.dameRegistroPorNombre ("Pedrito");
if (reg == null) {
    c.imprimeln ("No existe Pedrito en la Base de Datos.");
} else {
    c.imprimeln ("El registro es:\n"+reg);
}
```

La condicional **if** recibe una expresión booleana. No recibe ningún otro tipo de expresión; ni enteros ni flotantes ni caracteres. Sólo recibe booleanos. Si el valor de la expresión que recibe es **true**, se ejecuta el bloque. Si es **false**, se ejecuta el bloque del **else**.

Un **if** puede existir sin **else**, por ejemplo (estamos asumiendo ahora otro método que regresa una calificación):

```
float calificacion = p.dameCalificacion ();
c.imprimeln ("Tu calificación es: "+calificacion+".");
if (calificacion >= 8) {
    c.imprimeln ("Felicidades.");
}
```

La relación entre switch e if

El if es sólo una particularización del switch (o el segundo una generalización del primero, como prefieran). En general, podríamos usar solamente if; por ejemplo el primer ejemplo que vimos:

```
Consola c;
c = new Consola ();

Prueba p;
p = new Prueba ();

int estadoCivil = p.dameEstadoCivil ();

if (estadoCivil == Prueba.SOLTERO) {
    c.imprimeln ("Este␣sí␣es␣listo.");
} else if (estadoCivil == Prueba.CASADO) {
    c.imprimeln ("A␣éste␣ya␣lo␣agarraron.");
} else if (estadoCivil == Prueba.DIVORCIADO) {
    c.imprimeln ("Este␣recapacitó.");
} else if (estadoCivil == Prueba.VIUDO) {
    c.imprimeln ("Este␣tomó␣decisiones␣extremas.");
} else {
    c.imprimeln ("Este␣no␣quiere␣decir.");
}
```

En este ejemplo, XEmacs (y cualquier otro editor de código que se respete) indentará así los ifs. Pero hay que entender que el segundo if está anidado dentro del primero, y el tercero dentro del segundo, etc. Es por eso que el switch es mucho mejor para este tipo de casos. Pero existen ambas versiones para que se pueda utilizar la que más nos convenga, dependiendo de nuestro problema.

5.3.2. Iteraciones

¿Qué pasa si queremos imprimir los nombres de todos los registros en nuestra base de datos? Podemos hacerlo uno por uno, si suponemos que los conocemos todos a priori. O podemos utilizar iteraciones.

Las iteraciones nos sirven para repetir tareas un determinado número de veces, o hasta que alguna condición falle.

Los ciclos while y do ... while

En el caso de nuestra Base de Datos para agenda, si no supiéramos cuántos registros tiene, para imprimir los nombres de todos lo mejor sería usar un while. He aquí un ejemplo

```
int i = 0;

// Tenemos que inicializarlo porque lo usamos en el while.
String reg = "";
```

```
String nombre;

while (reg != null) {
    reg = bdd.dameRegistro (i);
    nombre = bdd.dameNombreDeRegistro (reg);
    c.imprimeln ("Nombre:␣"+nombre);
    i++;
}
```

(Aún no escribimos las funciones dameRegistro y dameNombreDeRegistro, pero no es muy difícil y deberían poder imaginarse cómo hacerlo.)

La iteración (o ciclo) **while** ejecuta su cuerpo mientras la condición que recibe sea verdadera. La expresión que recibe el **while** tiene que ser booleana.

Su estructura de control hermana, **do** ... **while** funciona de forma casi idéntica:

```
int i = 0;
String reg;
String nombre;

do {
    reg = bdd.dameRegistro (i);
    nombre = bdd.dameNombreDeRegistro (reg);
    c.imprimeln ("Nombre:␣"+nombre);
    i++;
} while (reg != null);
```

La única diferencia es que **while** comprueba primero la condición, y luego ejecuta el cuerpo. En cambio **do** ... **while** primero ejecuta el cuerpo y luego comprueba la condición; esto quiere decir que el **do** ... **while** ejecuta siempre su cuerpo al menos una vez.

El **while** y el **do** ... **while** son especialmente útiles cuando no sabemos de antemano cuántas veces vamos a iterar. Sin embargo, a veces sí sabemos exactamente cuántas veces vamos a iterar, y en estos casos se suele usar el **for**.

La iteración for

El **for** está pensado para iterar sobre rangos. Por ejemplo, para imprimir todos los nombres de nuestra Base de Datos para agenda haríamos:

```
int i;

// Con esto, calculamos el número de registros.
int numeroDeRegistros = tabla.length ()/TAM_REGISTRO;
String reg;
String nombre;

for (i = 0; i < numeroDeRegistros; i++) {
    reg = bdd.dameRegistro (i);
    nombre = bdd.dameNombreDeRegistro (reg);
    c.imprimeln ("Nombre:␣"+nombre);
```

```
}
```

La iteración **for** recibe tres expresiones. La primera se ejecuta incondicionalmente, pero sólo una vez. Generalmente se utiliza para inicializar variables que se usarán para recorrer algún rango. Esta primera expresión no necesita regresar valor.

La segunda expresión es booleana. Mientras esta expresión se cumpla (o sea, regrese **true**), el cuerpo del **for** se ejecutará. Tiene que regresar obligatoriamente un valor de tipo booleano.

La tercera expresión se utiliza generalmente para actualizar las variables inicializadas en la primera expresión. Tampoco necesita regresar ningún valor.

Al entrar al **for**, se ejecuta primero la inicialización, y se comprueba que la condición booleana sea verdadera. Si no es verdadera, no se ejecuta el cuerpo nunca.

Si sí es verdadera, se ejecuta el cuerpo del **for**, y al final se ejecuta el tercer argumento. Después vuelve a comprobar la condición booleana (la inicialización ya no vuelve a considerarse), y si es verdadera se repite todo.

El **for**, el **while** y **do ... while** son exactamente lo mismo. Un **while** puede escribirse con un **for**, y un **for** con un **while**; y si sólo se tuviera **while** o **do ... while** se podría emular al otro.

De hecho, hay lenguajes de programación donde sólo se da **while** o **do ... while**, no ambos. También hay lenguajes donde no se da el **for**.

Tener los tres nos permite mayor versatilidad al escribir código, y es bueno utilizar el que más se acomode en cada caso.

5.3.3. Las instrucciones break, return y continue

La instrucción **break** nos sirve para *romper* la ejecución de un **switch**, como ya vimos. Pero sirve también para romper cualquier bloque o cuerpo de un ciclo. Con el **break** podemos romper la ejecución de un **switch**, de un **while**, de un **do .. while** y de un **for**. La instrucción **continue** en cambio se salta lo que reste de la vuelta actual, para empezar inmediatamente la siguiente.

Habrá veces dentro de un método en que no sólo querremos salir de un **switch** o un ciclo, sino que querremos salir de toda la función. Para tales casos, hay que usar la instrucción **return**. Podemos usar la instrucción **return** en cualquier método, inclusive aquellos que tienen tipo de regreso **void**; en estos casos se utiliza el **return** solo, sin ninguna expresión a la derecha.

5.3.4. Listas

Las condicionales e iteraciones nos permiten tomar decisiones dentro de nuestro programa, lo que hace que el flujo de ejecución sea tan complejo como deseemos.

Ahora que las tenemos, es posible utilizar clases que nos permiten representar de forma más clara la información de un programa.

Las listas son un tipo abstracto de datos que nos permite tener un número arbitrario de algún tipo de elementos. Definiremos las listas como sigue

Una *lista* es

- **null** (si la lista es vacía), o
- un elemento seguido de una lista.

La definición es *recursiva*; hace referencia a lo que está definiendo. Piensen en los números naturales y los axiomas de Peano; un número natural es 1, o su antecesor (que es un número natural) más 1.[1]

Por su misma definición, las listas son divertidas para trabajar. Si una lista no tiene elementos, entonces decimos que es vacía y será sencillamente **null**. Si no es vacía, entonces tiene al menos un elemento, y además a su primer elemento le sigue una lista (con cero o más elementos).

Más adelante trabajaremos con listas de muchos tipos; por ahora utilizaremos sólo listas de cadenas utilizando la clase ListaDeCadena.

Actividad 5.1 Con asistencia de tu ayudante baja el archivo `util1.jar`, para que puedas utilizar la clase ListaDeCadena. También consulta la documentación generada de la clase.

Si una lista no es vacía, tiene dos partes: su elemento, que llamaremos *cabeza* y al que podremos tener acceso con el método dameElemento; y otra lista, a la cual le llamaremos la *siguiente* lista, y a la que podremos tener acceso con el método siguiente:

```
ListaDeCadena a;
a = p.llenaLaLista (); // Llena la lista de alguna manera.
if (a != null) {
    c.imprimeln ("Este␣es␣el␣primer␣elemento:␣"+a.dameElemento ());
    a = a.siguiente ();
    if (a != null) {
        c.imprimeln ("Este␣es␣el␣siguiente␣elemento:␣" +
                     a.dameElemento ());
    }
}
```

Vean el código de arriba; ahí perdimos al primer elemento. Al momento de hacer

```
a = a.siguiente ();
```

la única referencia que apuntaba al primer elemento de la lista (a), hacemos ahora que apunte al siguiente elemento. Ya no hay referencias apuntando al primer elemento, ya no podemos usarlo. Eso significa que queda marcado como removible, y que el recolector de basura se lo llevará. Y ya no hay manera de recuperarlo.

Con las listas, si tenemos a un elemento, podemos tener el que le sigue, pero no al *anterior*. Esto es importante; *no pierdan nunca la cabeza* (de las listas, por supuesto).

[1] Sí nos acordamos de los axioma de Peano, ¿verdad?

¿Cuántos elementos tiene una lista? Es muy fácil saberlo:

```
ListaDeCadena a;
a = p.llenaLaLista ();
ListaDeCadena b = a; // Para no perder la cabeza...
int contador = 0;
while (b != null) {
    b = b.siguiente ();
    contador++;
}
```

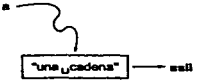
Vean que ahí no perdemos la cabeza (utilizamos a la variable de referencia b para movernos por la lista). El tamaño de la lista queda en la variable contador. La propia clase ListaDeCadena ofrece un método para saber cuántos elementos tiene; busca cuál es en la documentación generada de la clase.

El **while** del ejemplo anterior es muy importante, y será usado mucho con listas. Con un **while** de ese estilo podemos recorrer toda la lista, o recorrerla hasta encontrar algún elemento; y entonces romper el **while** con un **break** o con un **return**. Por supuesto, también funciona con un **do** ... **while** o con un **for**.

Hemos trabajado con una función imaginaria llenaLaLista, pero para crear una lista con un solo elemento solamente tendremos que usar **new**.

```
ListaDeCadena a;
a = new ListaDeCadena ("una cadena");
```

Esto crea una lista de la forma:

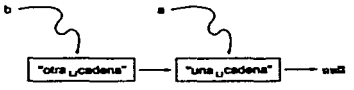


Noten que el constructor recibe una cadena, que es justamente el elemento de nuestra lista. Así construida, la lista que le sigue a a es **null**.

Ahora, como una lista tiene otra lista (la siguiente), también podremos construir listas a partir de listas:

```
ListaDeCadena a;
a = new ListaDeCadena ("una cadena");
ListaDeCadena b;
b = new ListaDeCadena ("otra cadena", a);
```

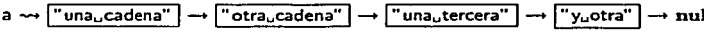
Esto quiere decir que tenemos dos constructores; uno recibe una cadena, y el otro recibe una cadena y una lista. El ejemplo de arriba crea una lista de la forma:



De esta forma, ahora a es el elemento siguiente de b. Noten que de esta manera las listas se construyen *al revés*; el último elemento en crearse se vuelve la cabeza. Para que no sea así, tendremos la función agrega, de la clase ListaDeCadena que hará justamente eso, agregar elementos al final de la lista:

```
ListaDeCadena a;
a = new ListaDeCadena ("una␣cadena");
a.agrega ("otra␣cadena");
a.agrega ("una␣tercera");
a.agrega ("y␣otra");
```

Esto último se vería así



Utilizaremos las listas mucho a lo largo de ésta y las siguientes prácticas.

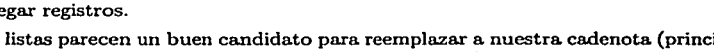
5.4. Ejercicios

1. Nuestra Base de Datos para agenda tiene una fuerte limitante: es estática. Para acentuar esto, la variable tabla de la clase es final; siempre es la misma. No podemos borrar o agregar registros.

   Las listas parecen un buen candidato para reemplazar a nuestra cadenota (principalmente por el hecho de que no hemos visto ninguna otra cosa).

   Haz que la clase BaseDeDatosAgenda utilice una ListaDeCadena en lugar de una cadena enorme. Cada registro será igual que antes; una cadena de tamaño TAM_REGISTRO.

   Queremos que nuestra Base de Datos tenga los mismos registros de la clase anterior, así que tendrás que crear un constructor para añadir los primeros registros (que puedes copiar de la práctica anterior). Nuestra antigua tabla debe quedar ahora como la lista:

   Ya tienes el archivo `util1.jar`. A estos archivos se les conoce como *jarfiles*, y sirven para guardar dentro de ellos clases y paquetes. Dentro de `util1.jar` está la clase ListaDeCadena. Para compilar un programa que la utilice deberás usar la línea de compilación

   ```
   # javac -classpath interfaz1.jar:util1.jar:. <TusClases.java>
   ```

   Nota que a la opción `-classpath` podemos darle varios *jarfiles*, separados por dos puntos (`:`). La clase ListaDeCadena está en un paquete llamado `icc1_1.util`, así que para utilizarla debes poner al inicio de las clases que lo usen la línea

   ```
   import icc1_1.util.ListaDeCadena;
   ```

   Para ejecutar un programa, ahora deberás utilizar la siguiente línea

```
# java -classpath interfaz1.jar:util1.jar:. <NombrePrograma>
```

2. Reescribe las funciones dameRegistroPorNombre y dameRegistroPorTelefono para que funcionen con la nueva implementación de listas. Vas a necesitar una iteración, porque ya no podemos utilizar los métodos de la clase String para buscar (aunque sí los vamos a necesitar para comparar cadenas). Prueba que los métodos funcionen en tu clase de uso.

   Observa que aunque hay que escribir bastante código en la clase BaseDeDatosAgenda, a la clase de uso no hay que hacerle nada. Esto es porque las *interfaces* del programa (sus métodos públicos) no han cambiado. Sí han cambiado por dentro; pero no han cambiado en nombre, parámetros o tipo de regreso. Por lo tanto para todas las clases que los usaban, es como si no hubieran cambiado.

3. Implementa la función pública agregaRegistro, que recibe una cadena con un registro, y regresa void.

   La función debe comprobar que el registro sea válido (i.e. que tenga longitud TAM_REGISTRO) y, si es así, añadirlo a la Base de Datos (o sea, a la lista). Comenta la función para JavaDoc.

   Ten en cuenta de que por ahora estamos poniendo dentro del constructor de la clase varios registros; entonces cuando sea llamado este método, la variable de clase lista ya no será vacía. Tienes que cubrir dentro del método el caso cuando la variable lista sea vacía (o sea, null).

5.5. Preguntas

1. Ahora que podemos agregar registros a nuestra Base de Datos de agenda, ¿qué otras cosas crees que le hagan falta?

2. ¿Cómo crees que estén implementadas las listas?

3. ¿Crees que podrías escribir la clase MiListaDeCadena, e implementar las funciones dameElemento, siguiente y agrega?

4. ¿Cómo lo harías? Sólo platícalo.

Herencia

A system software crash will cause hardware to act strangely and the programmers will blame the customer engineer.

– Murphy's Laws of Computer Programming #6

Semana:	Octava semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Herencia.
Objetivo:	Aprender a usar herencia.

Esta práctica se realiza durante la octava semana del curso. El profesor debe haber comenzado ya herencia, y seguramente seguirá viéndola a lo largo de la práctica.

La herencia es tal vez el concepto individual más poderoso de la Orientación a Objetos. Por lo mismo esta práctica es larga; no tanto como la cuarta, pero larga al fin y al cabo.

Java evita la herencia múltiple de una manera muy elegante, y permite atacar la herencia simple de una forma sencilla sin por ello perder nada de poder.

Java facilita mucho el comprender la herencia, ya que está firmemente ligada al diseño mismo del lenguaje, pero eso no impide que el tema tenga sus dificultades.

Los ejercicios son bastante más complejos (o entretenidos) que los vistos hasta ahora; pero se le proporciona al alumno suficientes pistas en la redacción de los mismos y en toda la práctica para que los lleve a cabo. Una semana debería bastarles a los estudiantes para completar la práctica.

Después de herencia, el único concepto fundamental de Java que se sigue evitando son la excepciones. Sin embargo, no veremos todavía excepciones, porque creemos que los siguientes conceptos que atacaremos (entrada/salida, arreglos y recursión) necesitan verse sin el ruido que pueden ocasionar las excepciones.

Práctica 6
Herencia

6.1. Meta

Que el alumno aprenda cómo funciona la herencia en Java.

6.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- comprender y utilizar la herencia en Java,
- entender qué es la jerarquía de clases,
- hacer conversión explícita de tipos, y
- entender las interfaces.

6.3. Desarrollo

La herencia es una de las herramientas más poderosas de la Orientación a Objetos. Nos permite reutilizar trabajo ya hecho, cambiar las interfaces de una clase sin por ello inutilizar otras clases que las usan, resolver varios problemas a la vez, y muchas cosas más.

La herencia es fundamental en el diseño Orientado a Objetos. Si el lenguaje otorga la facilidad de herencia y no se utiliza, se está desperdiciando gran parte del poder que se obtiene al utilizar un lenguaje Orientado a Objetos.

6.3.1. Heredar y abstraer clases

Hemos trabajado con la clase BaseDeDatosAgenda, que nos sirve para guardar registros de una agenda. ¿Qué sucede si nos piden ahora una Base de Datos para una nómina, o una librería?

Una vez que hayamos terminado la clase BaseDeDatosAgenda, nos sería muy sencillo hacer la clase BaseDeDatosLibreria, que haría en esencia *exactamente* lo mismo que la clase BaseDeDatosAgenda, pero en lugar de usar registros con nombres, direcciones y teléfonos, utilizaría registros con títulos, autores y números ISBN.

Sin embargo, esto nos llevaría a repetir mucho código ya existente, y hemos visto que una de las metas de la Orientación a Objetos es justamente reutilizar la mayor cantidad de código posible.

La herencia es una de las soluciones para la reutilización de código. Cuando uno hereda una clase A en una clase B, los métodos y variables de clase no privados de la clase A pueden ser usados automáticamente por los objetos de la clase B. Los métodos y variables privados *no* se heredan; todos los demás sí.

Una vez que la clase B se haya heredado de la clase A, puede definir métodos y variables de clase propios. Entonces los objetos de la clase B podrán hacer uso de los métodos y variables de clase de la clase A, y además los que se definan en la clase B. Por eso también se dice que heredar una clase es *extenderla*, y ambos términos serán equivalentes en estas prácticas.

Además de definir métodos propios, la clase B puede *redefinir* o *sobrecargar* los métodos de la clase A. Veremos un poco más de eso más adelante.

Si asumimos que tenemos una clase Vehiculo, y quisiéramos extenderla con la clase Bicicleta, entonces sólo tenemos que hacer

```
public class Bicicleta extends Vehiculo {
    ...
```

Casi todas las clases de Java se pueden extender (en un momento veremos cuáles no). Podemos extender nuestra clase Matriz2x2, o nuestra clase Consola, por ejemplo.

Sin embargo, habrá veces que escribiremos una clase con el único propósito de extenderla. En otras palabras, en ocasiones escribiremos clases de las cuales no pensamos instanciar objetos, sino heredarlas para instanciar objetos de las clases herederas. A estas clases las llamaremos *abstractas*, ya que no se concretarán en objetos. Para que una clase sea abstracta, sólo tenemos que agregar **abstract** a la declaración de la clase:

```
public abstract class Vehiculo {
    ...
```

Cualquier clase puede ser abstracta; la palabra clave **abstract** sólo hace que sea imposible instanciar objetos de esa clase. Sin embargo, cuando hagamos clases abstractas será generalmente porque tendrán métodos abstractos, métodos que no tienen definido todavía un comportamiento. Si una clase tiene un método abstracto, la clase misma tiene que ser abstracta, de otra forma no va a compilar.

Un método abstracto no tiene cuerpo. Por ejemplo, el siguiente método podría estar definido dentro de la clase Vehiculo que acabamos de mostrar

```
public abstract boolean transporta (Paquete p);
```

(No se pongan quisquillosos con la aparición espontánea de clases como Paquete; esto es un ejemplo sencillo).

Un método sin cuerpo parece no tener mucho sentido; pero es tal vez el punto central de la herencia. La clase Vehiculo tiene un método abstracto llamado transporta; esto significa que *todas* las clases que extiendan a Vehiculo *tienen* que implementar el método transporta (a menos que sean abstractas también).

Con esto *forzamos* el comportamiento de todas las clases herederas de la clase Vehiculo. Si una clase extiende a Vehiculo (y no es abstracta), entonces podemos *garantizar* que tiene un

método que se llama transporta, que regresa un booleano, y que recibe un objeto de la clase Paquete como parámetro.

Pero, ¿por qué no escribimos el cuerpo de transporta? Si todas las clases que extiendan a Vehiculo heredan todos sus métodos no privados, ¿entonces por qué no de una vez escribimos transporta? La respuesta es que una bicicleta transporta paquetes muy distinto a como lo hace un automóvil.

La idea de tener un método abstracto es que todas las clases herederas tendrán al método, pero *cómo* funcionará va a depender de la clase que hereda. El ejemplo de una clase vehículo es muy simple, pero sirve para entender este punto. Tenemos toda la gama posible de vehículos (bicicletas, automóviles, patines, helicópteros), y todos ellos pueden transportar paquetes; pero cómo lo hacen funciona de manera distinta en todos. Entonces declaramos un método transporta abstracto en la clase abstracta Vehiculo, y dejemos que cada clase que la extienda (Bicicleta, Automovil, Patines, Helicoptero, etc.) la implemente de acuerdo a cómo funcione la clase.

En nuestro ejemplo de la clase Bicicleta, como la clase no es abstracta, *tiene que* implementar al método transporta. Si fuera abstracta podría no hacerlo; pero como no lo es está obligada a implementarlo. Para que la clase Bicicleta implemente el método transporta, debe usar la misma declaración que en su clase padre (la clase Vehiculo), pero sin **abstract**:

```
public boolean transporta (Paquete p) {
    /* Aquí implementamos el método. */
    ...
}
```

La declaración debe ser *exactamente igual* que la de la clase padre; debe llamarse igual el método, debe regresar el mismo tipo, y debe tener el mismo número de parámetros y con el mismo tipo y orden en que aparecen (el nombre de los parámetros puede cambiar). El acceso al método debe ser el mismo, aunque un cambio está permitido. Veremos más del acceso al heredar un poco más adelante. Al nombre de un método, a su tipo de regreso y a los tipos y orden de los parámetros se le conoce en conjunto como la *firma* de un método.

Para que una clase heredera implemente una versión propia de algún método de su clase padre, no es absolutamente necesario que el método sea abstracto. A esto se le llama *redefinir* o *sobrecargar* un método (*overloading* es el término usado en la literatura en inglés). Incluso se puede utilizar el método que queremos sobrecargar dentro del método sobrecargado, usando **super**:

```
public void metodo () {
    ...
    super.metodo ();
    ...
}
```

La referencia **super** se utiliza cuando queremos hacer referencia a la clase padre. Esto funciona porque los objetos de una clase heredera pueden comportarse como objetos de la clase padre. Hay que recordar eso siempre.

Si tenemos una clase, y no queremos que sus herederas puedan sobrecargar algún método, entonces podemos definir al método como final:

ESTA TESIS NO SALE
DE LA BIBLIOTECA

```
public final void metodo () {
    ...
}
```

Habíamos dejado pendiente la explicación de qué era un método final desde la práctica 4. El significado es muy distinto a las variables finales; está únicamente relacionado con la herencia. Si un método es final, ya no puede redefinirse o sobrecargarse: será el mismo para todas las clases herederas.

Dijimos un poco más arriba que casi todas las clases podían extenderse. Podemos evitar que una clase sea heredada si la hacemos final:

```
public final class ClaseNoHeredable {
    ...
```

Funciona de la misma manera que con los métodos finales; si una clase es final, sencillamente no puede ser heredada o extendida.

Hay que tener en cuenta una última cosa al heredar una clase, los constructores *nunca* se heredan. Un constructor no puede heredarse porque un constructor define cierto estado inicial, y éste siempre es concreto. Por lo mismo, no tiene sentido un constructor abstracto.

Sin embargo, sí podemos llamar a los constructores de nuestra clase padre dentro de los constructores de la clase heredera, utilizando super:

```
super ();          // Llamamos a un constructor sin parámetros.
super (a, b, c); // Llamamos a un constructor con parámetros.
```

Por supuesto, los parámetros de super deben coincidir con los parámetros de algún constructor de la clase padre. Sólo podemos usar super para llamar a los constructores de la clase padre dentro de los constructores de la clase heredera.

6.3.2. El acceso en la herencia

Java provee un acceso menos restrictivo que private, pero no tan abierto como public para clases que heredan a otras clases. El acceso se llama *protegido* (protected). Permite restringir el acceso a métodos y variables a sólo "miembros de la familia", o sea, únicamente a clases que hereden.

Esto es muy útil ya que para clases que no sean herederas el acceso es para motivos prácticos privado, preservando el encapsulamiento de datos y la modularidad de componentes. Además, permite a clases "emparentadas" compartir variables y métodos que no queremos mostrar a nadie más.

Cuando uno sobrecarga un método, o implementa un método abstracto, debe preservar el acceso tal y como lo definía la clase padre. La única excepción a esta regla, es cuando el método tiene acceso protegido. Si el método tiene acceso protegido en la clase padre, la clase que extiende puede cambiarle el acceso a público; pero es el único cambio permitido. Al revés es ilegal; no se puede transformar un método público en uno protegido. Y todas las otras combinaciones también son ilegales.

6.3.3. La jerarquía de clases

En nuestro pequeño ejemplo con la clase Vehiculo hicimos lo que se conoce como una *jerarquía de herencia*. La clase Vehiculo es una *superclase* de varias clases; esto suele representarse como en la figura 6.1.

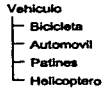


Figura 6.1: Jerarquía de clases

La jerarquía de clases tiene una gran importancia en Java; ordena a las clases en términos de comportamiento y características. La idea fundamental es que *todas* las clases de Java estén relacionadas en esta jerarquía, que toda clase sea heredera de alguna otra clase. Para garantizar esto, Java tiene una clase fundamental que se llama Object.

La clase Object es La Superclase (con mayúsculas). Todas las clases de Java son herederas de la clase Object en algún grado (lo que quiere decir que si no son herederas directas, su clase padre sí lo es, o si no la clase padre de la clase padre, etc.) Si una clase no extiende explícitamente a alguna otra clase, por omisión extiende a la clase Object. La clase Matriz2x2 es heredera directa de la clase Object por ejemplo, igual que las clases Reloj y Vehiculo.

Una de las principales ventajas que ofrece la herencia es que un objeto de una clase heredera garantiza que se puede comportar como un objeto de su clase padre. Un objeto de la clase Bicicleta se puede comportar igual que un objeto de la clase Vehiculo, porque tiene que implementar sus métodos abstractos, y porque hereda todos los métodos no privados (que son los que definen el comportamiento). Esto permite cosas como el siguiente ejemplo (es una clase de uso):

```
public static void main (String[] args) {
    Vehiculo v;
    v = instancia (obtenTipo ()); // Obtenemos algún tipo...
    if (v != null) {
        Paquete p;
        p = obtenPaquete (); // Obtenemos un paquete...
        v.transporta (p);
    }
}

public static Vehiculo instancia (int tipo) {
    switch (tipo) {
      case TODOS:
        return new Helicoptero ();
      case AUTOMOVIL:
        return new Automovil ();
      case BICICLETA:
        return new Bicicleta ();
```

```
            case PATINES:
              return new Patines ();
            default:
              return null;
          }
    }
```

Lo que ocurre en el ejemplo es que instanciamos un objeto de alguna clase heredera de la clase Vehiculo, y transportamos con él algún paquete. No sabemos de qué clase es el objeto v en el método main (en el método instancia lo instanciamos como Helicoptero, Automovil, Bicicleta o Patines, dependiendo del parámetro que recibamos); pero sabemos que tiene un método transporta, y lo usamos.

Piensen en los enteros (int) y los enteros cortos (short). Podemos utilizar un entero corto para inicializar un entero, porque un entero *se puede comportar* como un entero corto. No es tanto que un entero de 16 bits "quepa" en uno de 32 bits; el hecho importante es que un entero puede emular el comportamiento de un entero corto porque el conjunto de valores que abarcan los enteros contiene al conjunto de valores que abarcan los enteros cortos.

De la misma manera, el conjunto de objetos y métodos que abarca la clase Bicicleta es mayor que el de la clase Vehiculo; por tanto puede emular el comportamiento de un objeto de la clase Vehiculo. Igual pasa con el resto de las clases que extienden a Vehiculo.

Esto es muy útil, ya que entonces podemos instanciar dinámicamente dentro del programa el objeto con la clase que nos convenga, o con la que contemos. En el ejemplo de arriba, si tenemos todas las clases disponibles, entonces instanciamos un helicóptero (porque es el más rápido). Pero si sólo hay disponible una bicicleta, pues ni modo, usamos una bicicleta. En el método main no nos importa de qué clase sea nuestro objeto; sabemos que es de alguna clase heredera de Vehiculo y que por lo tanto tiene un método que transporta paquetes. Sólo comprobamos que el objeto no sea null, lo que significaría que no tenemos ningún vehículo disponible.

Todas las clases de Java (habidas y por haber) se pueden dibujar en árbol para mostrar la jerarquía de clases completa. Todas están conectadas, en muchos casos sólo por la clase Object, pero en otros casos hay relaciones más fuertes.

Todo esto nos lleva a que *absolutamente todos* los objetos de Java pueden comportarse como objetos de la clase Object, y por lo tanto utilizar sus métodos.

Actividad 6.1 Con el auxilio de tu ayudante, consulta la documentación de la clase Object. Ésos son métodos a los que *todas* las clases de Java tienen acceso.

Los 11 métodos de la clase Object tienen un significado especial cuando usamos Java. En particular, el método toString es el que utiliza Java para obtener la representación como cadena de un objeto.

La clase Object ofrece el método toString, por lo que todos los objetos de Java pueden llamarlo. En la clase Object el método toString está implementado de tal forma que regresa el nombre de la clase concatenado con un "@", concatenado con un entero que Java utiliza internamente para identificar unívocamente al objeto (pueden pensar que es la dirección de memoria donde vive el objeto, pero no siempre es así).

Por supuesto, podemos sobrecargar el método. En la clase Matriz2x2, por ejemplo, lo sobrecargamos para que pudiéramos pasar los objetos de la clase como cadenas al método imprime de la clase Consola.

Como todos los objetos de Java se pueden comportar como objetos de la clase Object, podemos hacer ahora una clase Lista cuyos elementos no sean cadenas, sino objetos de la clase Object. Con esto automáticamente ya tenemos listas que pueden contener objetos de todas las clases de Java.

Actividad 6.2 Con la asistencia de tu ayudante, consulta la documentación de la clase Lista. Observa que para motivos prácticos funciona de manera idéntica a la clase ListaDeCadena, nada más que recibe y regresa objetos de la clase Object en lugar de cadenas.
La clase Lista también está en el *jarfile* `util1.jar`, así que no tienes que bajar un nuevo archivo, pero sí tienes que importar a la clase usando **import** `icc1_1.util.Lista;`.

La clase Lista puede tener como elementos objetos de cualquier clase. Esto en particular significa que puede tener como elemento un objeto de la clase Lista. O sea, podemos tener listas de listas. Y listas de listas de listas.

Se puede complicar tanto como queramos.

6.3.4. Tipos básicos como objetos

Los tipos básicos de Java (**byte**, **short**, **int**, **long**, **float**, **double**, **char** y **boolean**) no son objetos; por lo tanto no forman parte de la jerarquía de herencia de Java. ¿Como hacemos una lista de enteros entonces?

Para incluir a sus tipos básicos dentro de la jerarquía de clases, Java ofrece las clases Byte, Short, Integer, Long, Float, Double, Character y Boolean, que son clases que "encapsulan" a los tipos básicos.

Para crear un objeto de la clase Integer que represente el valor de 5, utilizamos

```
Integer i;
i = new Integer (5);
```

El objeto i ahora puede ser agregado a una lista

```
Lista l;
l  = new Lista (i);
```

Para obtener el valor entero de este objeto, utilizamos el método intValue de la clase Integer:

```
int j = i.intValue();
```

Debe ser obvio que la función equivalente para la clase Float es floatValue. Estas clases no sólo envuelven a los tipos básicos de Java para que sean tratados como objetos; también ofrecen funciones que resultan útiles para el tipo básico al que representan y variables que nos dan información sobre el tipo (como MAX_VALUE de la clase Integer, que contiene un valor de tipo **int** que corresponde al entero de mayor tamaño que podemos representar con 32 bits).

Actividad 6.3 Con el apoyo de tu ayudante, consulta la documentación de las clases Byte, Short, Integer, Long, Float, Double, Character y Boolean.

6.3.5. Conversión explícita de tipos

Un poco más arriba vimos que le pasábamos un objeto de la clase Integer al constructor de la clase Lista

```
Lista lista;
lista = new Lista (i);
```

El constructor de la clase Lista recibe un objeto de la clase Object. Ya vimos que esto es válido porque al ser Integer una clase heredera de la clase Object, entonces puede comportarse como un objeto de la misma. La fuerte tipificación de Java no se rompe cuando vamos de una clase más particular a una más general. Sin embargo, al revés no funciona.

Cuando queramos recuperar el objeto de la clase Integer de la lista, no podremos hacer

```
Integer k;
k = lista.dameElemento();
```

Esto no va a funcionar porque dameElemento regresa un elemento de la clase Object, y no podemos garantizar que funcione como un objeto de la clase Integer. La clase Integer es más particular que la clase Object, y un objeto del que solamente se sabe que es de la clase Object no puede garantizar tener todos los métodos y características de un objeto de la clase Integer.

Pero nosotros *sí sabemos* que es de la clase Integer. Para poder hacer una *conversión explícita de tipos* (o *casting* en la jerga computacional), precedemos a la expresión que deseamos convertir con el tipo que queremos entre paréntesis

```
Integer k;
k = (Integer)lista.dameElemento();
```

Con esto le ponemos *máscara* de Integer al objeto de la clase Object. En el ejemplo que estamos manejando va a funcionar porque nosotros sabemos de qué tipo es el objeto que nos regresa la lista; pero si le hacemos una conversión a un objeto que no es del tipo al que estamos convirtiendo, la JVM marcará un error y terminará la ejecución del programa.

La conversión explícita de datos no sólo funciona con objetos; si queremos asignarle a un int el valor de un float hacemos

```
float f = 1.2F;
int   i = (int)f;
```

Habrá veces en que no sabremos con tanta seguridad de qué clase es un objeto. Por suerte, los objetos siempre recuerdan de qué tipo son, y Java ofrece un operador para comprobar la clase

de una referencia; **instanceof**, un operador binario *no* conmutativo cuyo operando izquierdo es una variable de referencia y el derecho el nombre de una clase:

```
Consola c;
c = new Consola ("Transporte de paquetes");
if (v instanceof Bicicleta) {
    c.imprimeln("Usamos una bicicleta para transportar.");
} else if (v instanceof Automovil) {
    c.imprimeln("Usamos un automóvil para transportar.");
}
...
```

Estamos volviendo a usar nuestro ejemplo de la clase Vehiculo. El operador **instanceof** regresa **true** si el objeto del lado izquierdo es instancia de la clase del lado derecho, y **false** en cualquier otro caso.

Si tienen dudas de cuándo hacer una conversión no se preocupen; el compilador no les va a permitir compilar si la conversión es necesaria. Pero una manera sencilla de plantearlo es imaginar el árbol de la jerarquía de clases; si van de abajo hacia arriba, no es necesaria la conversión explícita. Si van de arriba hacia abajo, sí es necesaria.

6.3.6. Interfaces

¿Qué pasa si queremos una clase que combine el comportamiento de otras dos, o tres clases? Lo lógico sería pensar en heredar todas las clases que queramos en nuestra clase. A esto se le llama *herencia múltiple*.

Sin embargo, Java no soporta herencia múltiple. Lo que Java ofrece a cambio son *interfaces*. Las interfaces son clases que no pueden implementar métodos; sólo pueden declararlos. Una interfaz es de esta forma

```
public interface Ejercitador {
    public void quemaCalorias (int kg);
}
```

El acceso de la interfaz tiene que ser público; no se admite ningún otro. Lo mismo ocurre con los métodos y las variables de clase. Y por supuesto, lo más importante es que los métodos no tienen cuerpo.

No podemos instanciar objetos de una interfaz. Las interfaces sirven para definir el comportamiento de una clase, sin implementar ninguno de sus métodos. Pero la principal ventaja es que una clase puede "heredar" varias interfaces al mismo tiempo.

Pusimos "heredar" (entre comillas) porque no es exactamente herencia, y de hecho tiene un término propio: *implementar*. Una clase puede implementar tantas interfaces como quiera, además de que puede extender a alguna otra clase (a lo más una). Si quisiéramos una clase que extendiera a la clase Vehiculo e implementara a la clase Ejercitador, haríamos:

```
public class Bicicleta extends Vehiculo
                       implements Ejercitador {
    ...
```

Si la clase Bicicleta está declarada así, entonces debe implementar los métodos abstractos de la clase Vehiculo, y además tiene que implementar todos los métodos declarados en la interfaz Ejercitador.

Las interfaces sólo pueden tener variables de clase que sean públicas y finales. Pueden ser estáticas también (y de hecho casi siempre lo serán).

Una clase puede implementar tantas interfaces como quiera:

```
public class Z implements A, B, C, D, E {
    ...
```

Las interfaces son la alternativa de Java a la herencia múltiple. Se puede hacer conversión de tipos hacia una interfaz, y pueden ser recibidas como parámetros y regresadas por métodos. Java en particular utiliza mucho las interfaces para señalar que una clase es capaz de hacer ciertas cosas.

6.3.7. La herencia en el diseño

La herencia modifica significativamente el cómo diseñaremos la solución a un problema. Básicamente tendremos dos modos de hacer diseño teniendo a la herencia en mente:

1. Diseñar desde el principio una jerarquía de herencia con clases abstractas e interfaces, extendiéndolas e implementándolas de manera que resuelvan el problema.

2. Diseñar una solución sin herencia y, a la mitad, darnos cuenta de que ciertas clases comparten varias características y que podemos crear una o varias clases abstractas que al heredarlas nos darán una jerarquía de clases que nos resolverá el problema.

Por supuesto, la idea es que en algún punto ustedes sean capaces de diseñar un programa utilizando el primer método, aunque no es sencillo. Lo natural es utilizar el segundo método.

Lo usual será casi siempre que al estar a la mitad de su análisis se percaten de que hay posibilidad de utilizar la herencia en ciertas partes del problema, y entonces tendrán que ver qué tanto conviene hacerlo. Casi siempre convendrá.

El primer método requiere tener ya experiencia programando, para que seamos capaces de detectar *patrones* de problemas, muchos de los cuales tendrán una metodología para resolverlos que involucrará herencia. No es sencillo llegar a ese nivel; e incluso habiendo llegado a él, la mayoría de las veces ocurrirá que a la mitad del camino descubramos una nueva manera de utilizar la herencia en nuestro problema.

El diseño es la parte más importante cuando resolvemos un problema. Sin embargo, no es algo intocable; la principal directiva del diseño Orientado a Objetos es que una vez completada gran parte de la implementación de nuestro problema, habrá que regresar a ver qué partes del diseño están mal y cambiarlas de acuerdo a ello. Habrá incluso ocasiones en que todo un problema tendrá que rediseñarse desde el principio (y en esas ocasiones lo mejor es detectarlo lo más pronto que se pueda).

Por supuesto, hay que diseñar tratando de evitar que eso ocurra. Pero si el introducir la herencia a algún diseño va a facilitarnos mucho la vida, aun a costa de rediseñar casi todo de nuevo, no hay que dudar y hay que introducir la herencia. Casi siempre valdrá la pena.

6.4. Ejercicios

Como ya sabemos utilizar la herencia, es hora de rediseñar nuestra base de datos para aprovecharla. Ya que vamos a rediseñar, veamos qué otras cosas están mal en nuestra Base de Datos:

- Tenemos que cambiar de utilizar la clase ListaDeCadena a utilizar la clase Lista, para que dejemos de representar a nuestros registros como cadenas.
- Ya que podemos meter cualquier clase en nuestras listas, vamos a utilizar una clase para representar a nuestros registros.

Se te va a proporcionar un nuevo archivo `BaseDeDatosAgenda.java`, que extiende a la clase abstracta BaseDeDatos (cuyo archivo también se te proporcionará). También se te dará la clase abstracta Registro.

La idea es que en la clase BaseDeDatos definimos los métodos agregaRegistro y quitaRegistro, ya que todas las Bases de Datos del universo los usan, y no tiene sentido andarlos repitiendo. Es tarea tuya entender cómo funcionan los métodos.

En la clase Registro declaramos dos métodos; equals y toString. Ambos son sobrecargas (o redefiniciones) de los mismos métodos en la clase Object.

1. Implementa la clase RegistroAgenda, que es requerida por la clase BaseDeDatosAgenda.

   Debe extender a la clase Registro, implementar los métodos equals, toString y los que tú creas necesarios, y además declarar las variables de clase que representarán el nombre, la dirección y el teléfono. El teléfono *no* debe ser cadena, y todas las variables de clase deberán tener acceso privado.

   El método equals debe primero ver que el objeto que recibe no sea **null**. Si lo es, regresa false. Si no es **null**, debe ver que el objeto sea instancia de la clase RegistroAgenda. Si no lo es, regresa false. Si es instancia de RegistroAgenda, entonces debe comprobar que el nombre, la dirección y el teléfono de **this** (el objeto que llama al método) sean iguales que las del objeto que recibe como parámetro. Si lo son, regresa **true**; si no, regresa **false**. Recuerda: las cadenas se comparan *lexicográficamente*.

   El método toString debe armar una cadena que represente de forma "agradable" (lo que **tú** entiendas por agradable) al registro, y regresarla.

   Deberás también definir métodos para tener acceso y cambiar las variables de clase (las primeras acostúmbrate a ponerles prefijo dame, y a las segunda define). Básate en la clase Matriz2x2, si quieres.

   La clase sólo debe tener un constructor, que recibirá el nombre, la dirección y el teléfono (el teléfono que reciba *no* debe ser de tipo cadena).

   Todos los métodos y constructores que se te piden deben tener acceso público. Si creas métodos auxiliares, hazlos privados (no es necesario hacer métodos privados para resolver el ejercicio).

2. Redefine los métodos dameRegistroPorNombre y dameRegistroPorTelefono en la clase BaseDeDatosAgenda, para que se adapten a nuestro nuevo diseño (con herencia y listas de

objetos). Los métodos conservan el nombre y los parámetros; pero ahora deben regresar un RegistroAgenda.

Date cuenta de que ya no es necesario hacer agregaRegistro porque está definido en nuestra clase padre.

3. Reescribe la clase UsoBaseDeDatosAgenda para que cree un objeto de la clase BaseDeDatosAgenda, lea 3 registros que el usuario introducirá por el teclado, y haga una búsqueda por nombre que el usuario también dará por teclado. Deberá imprimirse en la consola si el nombre buscado se encontró o no.

   Para leer del teclado volveremos a usar la clase Consola. Si quieres leer una cadena, sólo tienes que hacer

```
Consola c;
c = new Consola ("Base de Datos");

String nombre;

nombre = c.leeString ("Dame una cadena:");
```

   Esto abre una ventana de diálogo donde el usuario puede escribir su cadena. Con el auxilio de tu ayudante, consulta la documentación de la clase Consola y consulta todos los métodos lee.

4. Todas las clases y métodos que implementes deben estar comentados para la documentación de JavaDoc. Debe quedar claro en la documentación qué devuelve una función o para qué es cada parámetro.

6.5. Preguntas

1. Las funciones agregaRegistro y quitaRegistro implementadas en el archivo `BaseDeDatos.java` utilizan sólo cosas que hemos visto en ésta y prácticas anteriores. ¿Hay algo que no entiendas?

2. ¿Qué te parece el diseño que está tomando la base de datos? ¿Crees que hay algún error en él?

3. Te habrás dado cuenta de que para crear una Base de Datos de una librería (o de lo que fuera), casi sólo se necesitaría extender la clase Registro a una nueva clase. ¿Cómo lo harías tú?

Entrada/Salida y Arreglos

Any given program, when running, is obsolete.

– Murphy's Laws of Computer Programming #7

Semana:	Novena semana.
Tiempo de entrega:	Dos semanas.
Prerrequisitos:	Arreglos, entrada/salida.
Objetivo:	Aprender a usar entrada, salida y arreglos.

Esta práctica se realiza durante la novena semana del curso. El profesor debe haber visto ya entrada y salida, y comenzar a ver arreglos.

La entrada y la salida es algo que se suele dejar de lado en muchos primeros cursos de computación. Se deja como algo obvio o no muy trascendente, cuando en el fondo muchos problemas que aparecen en la implementación de un programa tienen que ver con la interacción del usuario con el programa o del programa con algún dispositivo.

Creemos que explicarles a los estudiantes claramente qué es lo que ocurre al escribir en disco, además de proporcionarles herramientas para que sus programas puedan conservar información a través de ejecuciones distintas es algo fundamental para el curso. Los estudiantes deben ver que toda la información que manejan sus programas no se queda en el aire; que pueden guardarla y conservarla, e incluso usarla como entrada para otros programas.

Java tiene la biblioteca de clases exclusivas para entrada y salida tal vez más poderosa en existencia. Maneja archivos binarios, archivos de texto, conexiones de red y muchas cosas más.

Los alumnos de este curso están casi listos para ver entrada y salida tal cual la maneja Java, pero les falta todavía ver excepciones. Como se explicó en la práctica anterior, preferimos darles una versión sin excepciones para el manejo de entrada y salida ya que las excepciones pueden distraer la atención de los estudiantes del objetivo central de esta práctica, que es la conservación de información a través de ejecuciones distintas.

En cuanto a los arreglos, los hemos dejado hasta esta parte en el curso porque creemos que los alumnos podrán apreciarlos mejor si han utilizado listas antes. Los arreglos son en cierta medida versiones restringidas de listas, más fáciles y más rápidos que las listas en muchos casos, pero menos poderosos. Si los alumnos han utilizado listas antes de utilizar arreglos, creemos que los apreciarán mejor y los utilizarán sólo cuando en verdad conviene, contrario a lo que ocurre muchas veces cuando se ven primero arreglos, que la gente después le cuesta más adaptarse al dinamismo y versatilidad que dan las listas.

Práctica 7
Entrada/Salida y Arreglos

7.1. Meta

Que el alumno aprenda cómo funcionan la entrada/salida y los arreglos en Java.

7.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- entender lo que es entrada y salida,
- entender lo que es un flujo (*stream*),
- utilizar las clases del paquete icc_1.es para escribir y leer datos en disco, y
- entender y utilizar arreglos.

7.3. Desarrollo

7.3.1. Entrada y Salida (E/S)

Un programa (a menos que sea puramente teórico) recibe una cierta entrada. Nuestros programas hasta ahora han recibido su entrada de distintas maneras.

Al inicio utilizábamos datos estáticos. Si queríamos buscar el nombre "Juan Pérez" en nuestra Base de Datos, teníamos que compilarlo estáticamente en la función main

```
public static void main (String[] args) {
    BaseDeDatos bd;
    bd = new BaseDeDatosAgenda ();

    ... // Llenamos la Base de Datos.

    Consola c;
    c = new Consola ("Base de Datos de Agenda");

    Registro r;
```

```
        r = bd.dameRegistroPorNombre ("Juan Pérez");

        if (r == null) {
            c.imprimeln ("El nombre \"Juan Pérez\" no existe.");
        } else {
            c.imprimeln (r);
        }
    }
```

El siguiente paso fue leerlo por teclado, haciéndolo dinámico. Para esto, utilizamos las funciones leeXXX de la clase Consola.

```
    public static void main (String[] args) {
        BaseDeDatos bd;
        bd = new BaseDeDatosAgenda ();

        ... // Llenamos la Base de Datos.

        Consola c;
        c = new Consola ("Base de Datos de Agenda");

        String nombre = leeString ("Nombre a buscar:");

        Registro r;
        r = bd.dameRegistroPorNombre (nombre);

        if (r == null) {
            c.imprimeln ("El nombre \""+nombre+"\" no existe.");
        } else {
            c.imprimeln (r);
        }
    }
```

Para la salida hemos utilizado hasta ahora las funciones imprime e imprimeln de la clase Consola.

La entrada y la salida son aspectos fundamentales de la programación, y la mayor parte de los problemas que uno encuentra en el momento de hacer un programa medianamente complejo, tienen que ver con la comunicación entre el usuario y la computadora (teclado/monitor); con la comunicación entre el programa y algún dispositivo externo (disco, cinta, CD-ROM), o con la comunicación entre el programa y algún otro programa, posiblemente en otra máquina (red).

En esta práctica cubriremos la entrada y la salida con más detalle de lo que lo hemos hecho hasta ahora.

Flujos

Por detrás de los telones, la clase Consola utiliza cadenas para comunicarse con el usuario; las funciones imprime e imprimeln transforman cualquier tipo que se les pase en cadena y lo imprimen, y las funciones leeXXX en realidad leen una cadena que luego se trata de convertir al tipo deseado (por ejemplo leeInteger utiliza la función estática parseInt de la clase Integer).

Actividad 7.1 Con el apoyo de tu ayudante consulta la documentación de las funciones parseByte, parseShort, parseInt, parseLong, parseFloat y parseDouble, que están en las clases Byte, Short, Integer, Long, Float y Double respectivamente.

Esto es porque la clase Consola está diseñada para crear una comunicación con el usuario sencilla y rápida. Pero la mayor parte de la comunicación que hay en un programa (esto es, su entrada y salida), es sencillamente el transporte de bytes de un lugar para otro. Así sea leer datos de un archivo, escuchar un puerto de la red, o esperar entrada a través del teclado, todo se reduce a una serie de bytes uno detrás del otro.

A esta serie de bytes se le suele denominar *flujo* (*stream* en inglés), haciendo referencia a un flujo de agua. Hay flujos de los que podemos determinar fácilmente dónde empiezan y dónde terminan (como los archivos en un disco duro o un CD-ROM) y hay flujos que no tienen un principio o un fin determinados (como las estaciones de radio en Internet, que mandan bytes ininterrumpidamente con la señal de la estación).

Viéndolos como flujos, hay *flujos de entrada*, es decir, de los que recibimos bytes, y *flujos de salida*, o sea a los que les mandamos bytes. Un archivo en un CD-ROM es un flujo de entrada (no podemos mandarle información, sólo obtenerla). Cuando guardamos un archivo en XEmacs, utilizamos un flujo de salida al mandar información al disco duro. Pine o GNUS, o en general cualquier lector de correo electrónico utiliza flujos de entrada para recibir correos de su servidor, y flujos de salida para mandarle al servidor de correo.

Todos los flujos funcionan de la siguiente manera:

1. Se crea el flujo (se abre el archivo en disco, se hace la conexión en red, etc.)
2. Si el flujo es de entrada, se lee de él.
3. Si el flujo es de salida, se escribe en él.
4. Al terminar, se cierra el flujo. Esto es importante, ya que generalmente un flujo utiliza un recurso del sistema que es necesario liberar (por ejemplo, casi todos los sistemas operativos tienen un número máximo de archivos que pueden tener abiertos al mismo tiempo).

Hay flujos que pueden abrirse simultáneamente para lectura y escritura. Sin embargo, esto no necesariamente representa una gran ventaja, y no veremos ese tipo de flujos en la práctica.

Filtros

Podemos conectar los flujos con programas o funciones, de tal manera que se reciban bytes por un flujo de entrada, el programa o función le haga algo a estos bytes, y se vuelvan a enviar a algún otro lado utilizando un flujo de salida. A estos programas o funciones se les denomina *filtros*.

Un filtro muy común, por ejemplo, es uno que tome los bytes de entrada y le aplique algún algoritmo que los comprima para mandarlos como salida. Otro filtro puede descomprimir los bytes. Un tercero puede utilizar cifrado de datos para proteger la información; un cuarto puede descifrar esos datos para poder leerla.

Si subimos el nivel de abstracción, y ocultamos con ello el hecho de que trabajamos en el fondo con bytes, todo programa es un filtro. Todo programa recibe cierta entrada que en el fondo son bytes, pero que nosotros abstraemos en un nivel más alto utilizando para ello el lenguaje de programación. Manejamos esos bytes como cadenas, enteros, flotantes, clases. El programa manipula de cierta manera la entrada, y nos regresa una salida que de nuevo en el fondo son bytes, pero que con ayuda del lenguaje manipulamos como entidades menos crudas que los bytes.

En el ejemplo que hemos utilizado, la entrada es un nombre, que es una cadena, y la salida es un registro, que es una clase, que también podemos representar como cadena gracias al método toString.

Java es de los lenguajes de programación que existen más poderosos para el manejo de entrada y salida. Tiene un conjunto de clases enorme y facilidades de abstracción de datos que nos permiten manejar la entrada y la salida en un nivel muy alto de abstracción. Pero también permite la manipulación directa de bytes desde su más bajo nivel hasta conversión de clases en bytes y viceversa. Las clases que se encargan de manejar flujos en Java están en el paquete java.io.

Actividad 7.2 Con asistencia de tu ayudante, consulta la documentación del paquete java.io.

Si Java provee la capacidad de manipular la entrada y la salida en un nivel alto de abstracción, ¿para qué bajarnos de nivel manipulando bytes directamente? La respuesta es que conforme vamos subiendo de nivel, generalmente a cambio perdemos un poco de velocidad[1] y de control.

Subir de nivel siempre es mejor porque ayuda al encapsulamiento de datos, y a la modularidad de un programa. También hace más fácil leer y mantener el código. Sin embargo, no siempre podremos hacer eso, o a veces nos convendrá más trabajar directamente en bajo nivel.

Manejo de archivos

Trabajar con bytes es tedioso; un trabajo de bajo nivel. Un byte puede representar cualquier cosa; puede ser una variable del tipo **byte** de Java, o parte del tipo **char**, que utiliza dos bytes, que a su vez puede ser parte de una cadena, como también puede ser parte de un **float**, que utiliza cuatro bytes.

Cuando se trabaja a bajo nivel se pueden hacer muchas cosas, pero en general son más difíciles de hacer y hay que tener más cuidado.

Cuando se trabaja a bajo nivel, si queremos guardar cadenas en un flujo de salida (un archivo en disco duro por ejemplo), tenemos que convertirlas a bytes y entonces mandarla por el flujo. Si queremos guardar un **int**, tenemos que hacer lo mismo. Y si luego queremos utilizar un flujo de entrada para leer los datos, tenemos que hacer el camino a la inversa.

Lo que nosotros quisiéramos, son clases a las que les dijéramos "guarda esta cadena", y que después sólo necesitáramos decirles "dame la cadena que guardé".

[1] Con computadoras cada vez más veloces, y compiladores cada vez más inteligentes, esto es cada vez menos perceptible.

Tales clases existen, y están en el paquete icc1_1.es, y se llaman Entrada y Salida. Son abstractas, para que otras clases concretas implementen las funciones de acuerdo al dispositivo que se use. En particular, las clases EntradaDeDisco y SalidaADisco son para escribir y leer datos en disco duro.

Actividad 7.3 Con auxilio de tu ayudante, consulta la documentación de las clases del paquete icc1_1.es, especialmente la de las clases EntradaDeDisco y SalidaADisco.
El paquete icc1_1.es está en otro *jarfile*, el archivo `es1.jar`. Van a necesitarlo para poder compilar y ejecutar clases que utilicen el paquete icc1_1.es.

Las clases EntradaDeDisco y SalidaADisco están hechas pensando en facilidad de uso antes que cualquier otra cosa. La clase SalidaADisco escribe sobre archivos de texto. Si uno hace

```
SalidaADisco sad;
sad = new SalidaADisco ("archivo.txt");

sad.escribeInteger (12);
sad.escribeFloat   (1.4F);
sad.escribeString  ("hola mundo");
sad.escribeBoolean (true);

sad.cierra ();
```

obtendrá un archivo de texto llamado `archivo.txt` que puede ser modificado usando cualquier editor como XEmacs. Pueden comprobarlo haciendo lo siguiente

```
# cat archivo.txt
12
1.4
hola mundo
true
#
```

Los métodos que leen de la clase EntradaDeDisco leen una línea del archivo que abren, y tratan de convertirla al tipo solicitado. Para leer el contenido del archivo `archivo.txt` sólo habrá que hacer

```
EntradaDeDisco edd;
edd = new EntradaDeDisco ("archivo.txt");

int     i = edd.leeInteger ();
float   f = edd.leeFloat   ();
String  s = edd.leeString  ();
boolean b = edd.leeBoolean ();

edd.cierra ();
```

Como pueden ver, el orden importa: si lo primero que escriben es un **double** asegúrense de que un **double** sea lo primero que lean.

Las clases son muy sencillas de usar; cualquier error grave que ocurra (que no haya permisos para leer o escribir un archivo, por ejemplo), resultará en que el programa termine con un mensaje de error.

Actividad 7.4 Aun cuando las clases del paquete `icc1_1.es` son de alto nivel, el uso de entrada y salida siempre es complicado. Para comprender mejor su funcionamiento, baja el archivo `UsoEntradaSalidaDisco.java` y compila y ejecuta el programa:

```
# javac -classpath interfaz1.jar:util1.jar:es1.jar:. \
        UsoEntradaSalidaDisco.java
# java -classpath interfaz1.jar:util1.jar:es1.jar:. \
        UsoEntradaSalidaDisco
```

Nota que el \ es para que puedas continuar una línea de comandos en varios renglones. Examina el código de la clase para que veas cómo funcionan las clases EntradaDeDisco y SalidaADisco.

7.3.2. Arreglos

Cuando hablamos de flujos, dijimos que se trataban de sucesiones de bytes, uno detrás del otro. Las listas son sucesiones de objetos.

Sin embargo, si sólo tenemos el primer elemento de una lista, o sea la cabeza, no podemos saber directamente, sin recorrer la lista, dónde está el n-ésimo.

En Java hay otra estructura que es una sucesión de objetos, o de tipos básicos de Java, pero con la ventaja de que están ocupando posiciones contiguas en la memoria. Como ocupan posiciones contiguas, si sabemos dónde está el primero podemos saber dónde está el segundo, y el tercero, y el n-ésimo. Podemos tener *acceso directo* a los objetos o tipos básicos. Esta estructura son los *arreglos*.

Las listas son una estructura de tamaño arbitrario. De entrada no sabemos cuántos elementos en total tiene una lista.

Esto es dinámico y muy provechoso; pero tiene la gran desventaja de que al momento de buscar un elemento, tenemos que recorrer gran parte de la lista para encontrarlo. En el peor de los casos (cuando el elemento que buscamos está al final de la lista) la recorremos toda.

A veces sabemos que vamos a necesitar *a lo más* k elementos. Para estos casos son los arreglos.

Los arreglos son sucesiones de objetos o tipos básicos de Java, ordenados uno después de otro, y a los que podemos tener acceso de forma directa gracias justamente a que están en posiciones contiguas en memoria (al contrario de las listas).

Para declarar un arreglo (por ejemplo de enteros), hacemos

```
int[] arreglo;
```

Los arreglos son objetos calificados de Java. Heredan a la clase **Object** y tienen acceso a todos los métodos de la clase. Ya que son objetos, hay que instanciarlos para poder usarlos. Los arreglos se instancian así:

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

Aquí instanciamos el arreglo para que tenga 5 elementos; pero el número de elementos puede ser arbitrariamente grande. Dentro de los corchetes en la instanciación podemos poner cualquier expresión de tipo **int**; pero si es un entero negativo, la JVM terminará la ejecución del programa al tratar de instanciar el arreglo.

Una vez definido cuántos elementos tendrá un arreglo, no podemos hacer que tenga más (contrario a las listas). Podemos crear otro arreglo más grande y copiar los elementos que el primero tenía, pero no podemos *agrandar* un arreglo.

Para tener acceso al n-ésimo elemento de un arreglo, sólo ponemos el nombre del arreglo, y entre corchetes el índice del elemento que queramos. Por ejemplo, para tener acceso al tercer elemento del arreglo de enteros que declaramos arriba hacemos:

```
arreglo[2];
```

Ése es el tercer elemento, ya que comenzamos a contar desde el cero. Por tanto, los elementos de un arreglo tienen índices que van desde 0 hasta el número de elementos menos uno. El elemento arreglo [2] es una variable de tipo **int** y podemos utilizarla como cualquier otra variable de tipo **int**:

```
arreglo[2] = 15;
c.imprimeln (arreglo[2]);
```

Cuando instanciamos al arreglo con **new**, el arreglo tiene todos sus elementos sin inicializar, pero Java les asigna 0 por omisión. Si fuera un arreglo de objetos, los inicializaría como **null**; si fuera de booleanos como **false**, etc.

Podemos pasar arreglos como parámetros:

```
public void promedio (int[] arreglo) {
    ...
```

y regresarlos también:

```
public int[] dameArreglo () {
    ...
```

En estos casos necesitamos saber el tamaño del arreglo que nos pasan o que nos regresan. Todos los arreglos (recuerden que son objetos) tienen una variable pública y final llamada **length** que nos dice cuántos elementos tiene el arreglo:

```
int tam = arreglo.length;
c.imprimeln ("El␣arreglo␣tiene␣"+tam+"␣elementos.");
```

La variable es final, para que no podamos cambiar su valor.

No sólo podemos hacer arreglos de tipos básicos; podemos hacer arreglos de cualquier tipo de Java, incluyendo por supuesto cualquier clase:

```
RegistroAgenda[] registros;
registros = new RegistroAgenda[1000];
```

Es importante recordar que registros [0], registros [1], registros [2], ..., registros [999] son referencias inicializadas con **null**, o sea, registros [0] es igual a **null**, registros [1] es igual a **null**, etc.

Cuando instanciamos con **new** un arreglo, si es de objetos hay que instanciar a pie cada uno de los elementos del arreglo:

```
registros[0] = new RegistroAgenda ("José␣Arcadio␣Buendía",
                                   "Macondo", 00000001);
registros[1] = new RegistroAgenda ("Úrsula␣Buendía",
                                   "Macondo", 00000002);
...
registros[999] = new RegistroAgenda ("Remedios␣Buendía",
                                     "Macondo", 00000999);
```

La variable registros (sin corchetes) es una variable de referencia a un arreglo de objetos de la clase RegistroAgenda, y el recolector de basura sigue el mismo criterio de las demás referencias en Java para liberar la memoria que ocupa.

Varias dimensiones

A veces una simple sucesión de objetos no nos basta. Por ejemplo, si queremos representar matrices, necesitamos arreglos de dos dimensiones.

Para esto hacemos

```
int[][] theMatrix;
theMatrix = new int[7][5];
```

Con esto creamos una matriz de 7 por 5. De aquí es evidente cómo crear arreglos de tres dimensiones:

```
int[][][] theCube;
theCube = new int[7][5][6];
```

Podemos complicarlo tanto como queramos.

Arreglos al vuelo

Si queremos un arreglo con los cinco primeros números primos, tendríamos que hacer

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

primos[0] = 2;
primos[1] = 3;
```

```
primos[2] = 5;
primos[3] = 7;
primos[4] = 11;
```

Esto es tedioso. Si al momento de compilar sabemos qué valores tendrá un arreglo, podemos crearlo al vuelo

```
int[] primos = { 2, 3, 5, 7, 11 };
```

con lo que nos ahorramos mucho código. Por supuesto, se puede hacer esto con cualquier tipo de Java.

```
String[] cuates = { "Rafa", "Erick", "Yazmín", "Karina", "Citlali" };
```

Tampoco es necesario que los valores sean literales:

```
RegistroAgenda ra1 =
  new RegistroAgenda ("José␣Arcadio␣Buendía", "Macondo", 00000001);
RegistroAgenda ra2 =
  new RegistroAgenda ("Úrsula␣Buendía", "Macondo", 00000002);
RegistroAgenda ra3 =
  new RegistroAgenda ("Aureliano␣Buendía", "Macondo", 00000003);

RegistroAgenda[] registros = { ra1, ra2, ra3 };
```

Esta forma es muy cómoda, mas si el contenido o tamaño de un arreglo sólo se puede saber en tiempo de ejecución, tenemos que utilizar la primera forma de instanciación.

Los argumentos de main

Hemos utilizado la siguiente declaración para main

```
public static void main (String[] args) {
    ...
```

El método main *tiene* que ser declarado así. Si no es público, o no es estático, o no se llama main, o no recibe un arreglo de cadenas como parámetro, entonces *ése no es* el método main, el punto de entrada a nuestro programa.

Con arreglos ya podemos entender completamente al método main; es un método público (porque tiene que llamarse desde fuera de la clase, lo llama la JVM), es estático, porque no puede ser llamado por ningún objeto (es el primer método que se llama, ¿de dónde podríamos sacar un objeto para llamarlo?), no regresa nada (su tipo de regreso es void; ningún otro es permitido), por supuesto se llama main, y recibe un arreglo de cadenas como parámetro.

¿Por qué se le pasa un arreglo de cadenas a main? La razón es para pasarle argumentos a un programa. Por ejemplo, si queremos pasarle argumentos a UsoBaseDeDatosAgenda haríamos

```
# java -cp interfaz.jar:util.jar:es1.jar:. \
          UsoBaseDeDatosAgenda arg 123 "hola mundo"
```

En este caso, "arg", "123" y "hola mundo" serían los argumentos del programa. Dentro de main, tendríamos acceso a ellos con args [0], args [1] y args [2] respectivamente.

Hay que recordar que son cadenas; el segundo argumento se interpreta como la cadena "123", no como el entero 123. Ésta es la única manera de pasarle parámetros al método main.

7.4. Ejercicios

Nuestra Base de Datos de agenda tiene un grave problema, los registros que damos de alta durante la ejecución del programa dejan de existir cuando el programa termina. En ese momento desaparecen.

Los ejercicios de esta práctica consistirán en que la Base de Datos de agenda tenga memoria *permanente*, o sea, que sea capaz de guardar su estado en disco duro.

Para escribir en disco duro utilizaremos la clase SalidaADisco, heredera de la clase Salida. La documentación de la clase puedes consultarla con el auxilio del ayudante.

Utiliza la clase UsoEntradaSalidaDisco para que veas un ejemplo de cómo utilizar la clase SalidaADisco.

1. En la clase BaseDeDatos implementa el método guardaBaseDeDatos con la siguiente firma:

```
public void guardaBaseDeDatos (Salida sal) {
    ...
```

   Como puedes ver, la función es concreta. Con esto hacemos que todas las Bases de Datos que extiendan a BaseDeDatos puedan guardar sus registros. En particular, si el método funciona correctamente, nuestra clase BaseDeDatosAgenda podrá guardar sus registros sin necesidad de hacerle nada.

   El método debe guardar el número de registros que hay en la Base de Datos: la longitud de la lista. Después debe guardar los registros.

   Para guardar los registros, asume que la clase Registro tiene un método que se llama guardaRegistro que recibe un objeto heredero de la clase Salida. Gracias a ello, para guardar los registros sólo tendremos que recorrer la lista y pedirle a cada registro que se guarde.

   Si asumimos el método guardaRegistro, podemos hacer el método guardaBaseDeDatos en la clase abstracta, ya que no tenemos que saber nada de los registros para pedirles que se guarden. Sólo necesitamos saber que se pueden guardar.

   El objeto heredero de la clase Salida que recibe nuestro método como parámetro es un objeto *prestado*. No le corresponde al método guardaBaseDeDatos crearlo; lo recibe ya creado. Lo usamos para guardar el número de registros, y luego se lo pasamos al método guardaRegistro cuando guardemos cada uno de los registros, pero no lo construimos ni lo cerramos.

   El crear el objeto y cerrar el flujo se realizará *afuera* de este método.

2. En la clase Registro declara la función abstracta guardaRegistro de la siguiente manera

```
public abstract void guardaRegistro (Salida sal);
```

Con esto forzamos que todos los objetos de alguna clase heredera de Registro tendrán un método guardaRegistro. El método del primer ejercicio necesita eso.

Ahora implementa el método *concreto* guardaRegistro en la clase RegistroAgenda (tienes que hacerlo; si no lo haces la clase RegistroAgenda dejará de compilar).

Igual que el método guardaBaseDeDatos, guardaRegistro recibe *prestado* el objeto heredero de la clase Salida que le pasamos como parámetro. No crearemos dentro del método al objeto, ni invocaremos su método cierra.

Dentro de guardaRegistro sólo guardaremos los campos del registro, usando para ello al objeto heredero de la clase Salida. Eso es lo único que hace el método.

3. Implementa el método estático recuperaRegistro en la clase RegistroAgenda con la siguiente firma

```
public static RegistroAgenda recuperaRegistro (Entrada ent) {
    ...
```

¿Por qué estático? Porque vamos a *recuperar* un registro; no podemos utilizar un objeto de la clase RegistroAgenda para que se recupere a sí mismo. Eso no tiene sentido; por eso usamos un método estático, para que la clase misma recupere el registro.

El método recibe un objeto heredero de la clase Entrada; al igual que su método hermano guardaRegistro, el objeto es prestado. Ni lo creamos ni lo cerramos dentro de recuperaRegistro.

No podemos hacer un método abstracto recuperaRegistro en la clase Registro (desarrollen un momento la lógica en un método estático y abstracto y verán por qué es imposible). Pero veremos en un momento que no es necesario.

El método recuperaRegistro debe recuperar los campos de un registro de agenda, instanciarlo con new, y regresarlo. Para recuperar los campos debe utilizar el objeto heredero de la clase Entrada; asegúrate de que respete el orden en que los guardó guardaRegistro.

4. Declara el método abstracto recuperaBaseDeDatos en la clase BaseDeDatos de la siguiente manera:

```
public abstract void recuperaBaseDeDatos (Entrada ent);
```

El método es abstracto porque como vimos en el ejercicio anterior, el método recuperaRegistro tiene que ser estático. Y no podríamos saber qué tipo de registro llamar en la clase BaseDeDatos, que es nuestra superclase de Bases de Datos. Mas no es necesario; basta que lo declaremos abstracto, y entonces con ello garantizamos que todas las Bases de Datos pueden recuperarse a sí mismas.

Por supuesto, tienen que implementar el método concreto recuperaBaseDeDatos en la clase BaseDeDatosAgenda con la siguiente firma:

```
public void recuperaBaseDeDatos (Entrada ent) {
    ...
```

Como su método hermano guardaBaseDeDatos, el objeto heredero de la clase Entrada no se crea ni se cierra dentro del método. Sólo lo toma prestado.

El método recuperaBaseDeDatos debe leer el número de registros (que es lo primero que debe guardar el método guardaBaseDeDatos), y después con un ciclo leer todos los registros utilizando el método recuperaRegistro de la clase RegistroAgenda. Cada registro recuperado deberá ser agregado a la Base de Datos.

Si se están preguntando por qué una Base de Datos sí puede recuperarse a sí misma, y por qué un registro no, la respuesta es que la Base de Datos puede vivir tranquilamente sin ningún registro, mientras que un registro no puede vivir sin campos. Los registros *son* sus campos.

5. Has usado los métodos leeString, leeInteger, leeFloat, etc. para obtener entrada del usuario por el teclado. Sin embargo es incómodo para el usuario tener que estar escribiendo respuestas una por una. Lo ideal sería que pudiera contestar varias preguntas de una vez.

   Para esto, la clase Consola ofrece las funciones hazPreguntas y dameRespuestaXXX. La primera recibe un arreglo de preguntas

```
String[] pregs = { "Nombre:", "Dirección:", "Teléfono" };
boolean resultado = c.hazPreguntas (pregs);
```

   con lo que una ventana de diálogo aparece donde se permite contestar todas las preguntas que estén en el arreglo. La ventana tiene un botón de "Aceptar", para que se guarden en memoria las respuestas del usuario, y otro botón de "Cancelar" para cancelar las preguntas. hazPreguntas regresa true si el primer botón es presionado, y false si se presiona el segundo o se cierra la ventana.

   Para obtener las respuestas, se utilizan las funciones dameRespuestaXXX, donde XXX es Integer, String, etc. Si se quiere obtener la primer respuesta como una cadena y la segunda como un entero se hace

```
String resp1 = c.dameRespuestaString (0);
int resp2 = c.dameRespuestaInt (1);
```

   Es responsabilidad del programador que si se hicieron n preguntas, no se pidan más de n respuestas. En el archivo `UsoEntradaSalidaDisco.java` puedes ver un ejemplo de cómo se utilizan los métodos.

   Con estos nuevos métodos, modifica UsoBaseDeDatosAgenda para que

   a) Cree una Base de Datos de agenda.

   b) Le pregunte al usuario cuántos registros desea introducir en la Base de Datos.

   c) Haga un arreglo de registros del tamaño que el usuario introdujo.

   d) Pida ese número de registros con un **for**, y que cada registro sea pedido usando las funciones hazPreguntas y dameRespuestaXXX. Si el usuario cancela el diálogo (o sea, el método hazPreguntas regresa **false**), el registro correspondiente en el arreglo debe inicializarse con **null**.

e) Cuando haya terminado de leer todos los registros en el arreglo, con otro **for** agregue los registros del arreglo en la Base de Datos. Debe comprobarse que los elementos del arreglo no sean **null**; si lo son, no deben agregarse en la Base de Datos.

f) Una vez agregados todos los registros no nulos, debe preguntarle al usuario un nombre de archivo, crear con él un objeto de la clase SalidaADisco, y con ese objeto guardar la Base de Datos. Hay que cerrar el flujo después de guardar la Base de Datos.

g) Crea *otro* objeto de la clase BaseDeDatosAgenda.

h) Después debe preguntar *de nuevo* por un nombre de archivo, y utilizar ese nombre para crear un objeto de la clase EntradaDeDisco, y usándolo recuperar la nueva Base de Datos. Cierra el flujo al acabar.

i) Pedir un nombre y buscarlo en la nueva Base de Datos recuperada.

Por su puesto, todo esto podría hacerse de una manera mucho menos rebuscada, pero queremos

- Que prueben todos los métodos que hagan.
- Hacerles más divertida la vida.

6. Por supuesto *todos* los métodos deben tener sus correspondientes comentarios para JavaDoc.

7.5. Preguntas

1. En esta práctica para guardar la Base de Datos sólo guardamos el número de registros y después los registros. ¿Se te ocurre una mejor manera de guardar la Base de Datos?
2. ¿Crees que sería mejor si la Base de Datos utilizara un arreglo en lugar de una lista?

Recursión

Any given program costs more and takes longer.

– Murphy's Laws of Computer Programming #8

Semana:	Undécima semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Recursión, pila de ejecución.
Objetivo:	Aprender a usar la recursión.

Esta práctica se realiza durante la decimoprimera semana del curso. El profesor debe haber visto ya recursión y la pila de ejecución de un programa.

La recursión es una de las herramientas más poderosas de la computación. Y Java, a pesar de ser imperativo y Orientado a Objetos, la maneja de una manera muy eficiente.

La práctica es muy sencilla, ya que el concepto de recursión no debería resultarles tan complejo a estudiantes que conocen los axiomas de Peano y que han utilizado listas durante ya varias semanas. El ejercicio también es corto, y la semana que se les da debería ser más que suficiente para realizarla.

Práctica 8
Recursión

8.1. Meta

Que el alumno aprenda a usar recursión.

8.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- entender y utilizar la recursión, y
- usar recursión en arreglos y listas.

8.3. Desarrollo

Podemos hacer llamadas de funciones dentro de otras funciones. Dentro del método main en nuestras clases de uso llamamos a varias funciones.

Dentro de esas mismas funciones podemos llamar a otras funciones. El método guardaBaseDeDatos de la clase BaseDeDatos llama al método guardaRegistro de la clase Registro.

Si podemos llamar funciones desde funciones, esto lleva a una pregunta casi inmediata; ¿qué pasa si llamamos a una función dentro de ella misma? Y la respuesta es la *recursión*.

8.3.1. Un mal ejemplo: factorial

La función factorial es el ejemplo clásico para mostrar la recursión, aunque es un mal ejemplo como veremos más adelante.

La función factorial (o $n!$) está definida para los números naturales (y el cero) de la siguiente manera

El factorial de un número $n \in \mathbb{N} \cup \{0\}$, que denotaremos como $n!$, es

- 1 si $n = 0$
- $n \cdot (n-1)!$ en otro caso.

¿Cómo definimos entonces la función factorial? Recursivamente es trivial (asumimos que el int n no es negativo):

```
int factorial (int n) {
    // Cláusula de escape.
    if (n == 0) {
        return 1;
    }

    return n*factorial (n-1);
}
```

Lo que hace entonces factorial es llamarse a sí misma con un valor distinto cada vez; ésa es la idea básica de la recursión. La comprobación que aparece en el método debe aparecer en toda función recursiva; se le denomina *cláusula de escape* o *caso base*, y nos permite *escapar* de la función.

Si una función recursiva no tiene cláusula de escape o está mal definida, nunca podremos salir de ella. El programa quedará atrapado para siempre en esa función, *y eso es algo que nunca debe ocurrir*. A quedar atrapado en una función recursiva se le llama muchas veces *caer en loop*.

Dijimos que la función factorial es un mal ejemplo de recursión. Esto es porque podemos hacer factorial iterativamente muy fácil

```
int factorial (int n) {
    int r;

    for (r = 1; n > 0; n--) {
        r = r*n;
    }

    return r;
}
```

En Java, una función recursiva será siempre (o en la mayoría de los casos) más cara (en recursos de la computadora) que una función no recursiva. La recursión afecta tanto en velocidad como en memoria.[1] Sin embargo, el enfoque recursivo es elegante en extremo, y puede simplificar el código de manera significativa.

En el caso de factorial, no tiene sentido hacerla recursiva. La versión iterativa es fácil de entender y no complica el algoritmo. Pero hay algoritmos en los que la versión iterativa es

[1] Otra vez: el aumento en la velocidad de las computadoras y la existencia de compiladores cada vez más inteligentes hacen esta diferencia cada vez menos perceptible. Mas la diferencia *está ahí*.

extremadamente compleja, y en estos casos siempre es mejor utilizar la recursión, como veremos enseguida.

8.3.2. Un buen ejemplo: las Torres de Hanoi

Tenemos el siguiente problema: tres postes, uno de los cuales tiene cierto número de discos de distintos diámetros ordenados de mayor a menor diámetro de abajo hacia arriba (ver la figura 8.1).

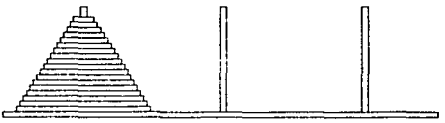


Figura 8.1: Torres de Hanoi

El reto es relativamente sencillo; tenemos que mover los discos del primer poste al segundo, con la siguiente restricción: no podemos poner nunca un disco sobre otro de diámetro menor.

¿Cómo movemos los n discos del primer al segundo poste? Recursivamente es sencillo; nuestro caso base (el que nos da la pauta para la cláusula de escape), es cuando sólo tenemos un disco. En este caso, solamente hay que mover el disco del primer poste al segundo poste.

Ahora sólo falta la recursión, que es cómo mover n discos.

La recursión es un poco más complicada, ya que tenemos la restricción de que no podemos poner un disco sobre otro de menor diámetro. Pero como comenzamos con los discos ordenados, sólo hay que encontrar cómo no romper el orden.

Esto resulta sencillo ya que tenemos tres postes; entonces para mover n discos del primer poste al segundo poste, primero movemos $n-1$ discos del primer poste al tercero, ayudándonos con el segundo, movemos el disco más grande del primer al segundo poste (ya sabemos cómo mover un solo disco), y después movemos de nuevo los $n-1$ discos del tercer poste al segundo, ayudándonos con el primero.

Y eso resuelve el problema:

```
/**
 * Pasa un disco del poste p1 al poste p2.
 */
public void pasa (Poste p1, Poste p2) {
    // Pasamos el disco del poste p1 al poste p2.
}

/**
 * Pasa n discos del poste p1 al poste p2, ayudándose
 * con el poste p3.
 */
public void pasa (Poste p1, Poste p2, Poste p3, int n) {
```

```
        if (n == 1) {
            Pasa (p1, p2);
        } else {
            Pasa (p1, p3, p2, n-1);
            Pasa (p1, p2);
            Pasa (p3, p2, p1, n-1);
        }
    }
```

Parece mágico, ¿verdad?

Actividad 8.1 Con la asistencia de tu ayudante, baja el *jarfile* `ejemplos1.jar`. El *jarfile* contiene varias clases con ejemplos; en particular tiene una muestra gráfica de las Torres de Hanoi. Ejecútala con la siguiente línea de comandos:

```
# java -classpath ejemplos1.jar:. icc1_1.ejemplos.Hanoi 5
```

El parámetro 5 es el número de discos a utilizar. Prueba con más discos, si no tienes nada mejor que hacer.

Puedes comprobar que el programa de ejemplo utiliza el algoritmo de arriba, paso por paso.

La recursión tiene mucho que ver con la inducción matemática. En la inducción matemática, sólo hay que ver un caso base, asumirlo para $n = k$ y demostrarlo para $n = k + 1$. El mismo principio se aplica aquí; decimos cómo manejar nuestro caso base (la cláusula de escape), asumimos que funciona para $n - 1$ y lo hacemos para n.

Las Torres de Hanoi es un algoritmo *doblemente recursivo*; hace dos llamadas a sí misma dentro de la función. El algoritmo iterativo de las Torres de Hanoi ocupa varias páginas de código, en comparación de las menos de quince líneas que utilizamos.

Hay una leyenda oriental que dice que si algún día alguien termina de jugar las Torres de Hanoi con 64 discos, el Universo dejará de existir. Sin embargo no hay que preocuparse; si se moviera una pieza por segundo, uno tardaría más de quinientos mil millones de años en terminar de jugar.

Las Torres de Hanoi es uno de los ejemplos de problemas que tienen solución, que la solución puede ser llevada a cabo por una computadora, y que sin embargo con una entrada no muy grande (64 en este caso) tardaría *mucho* tiempo en dar el resultado, haciéndolos para todo caso práctico intratables.

8.3.3. Listas y recursión

La definición de factorial y de muchos algoritmos que naturalmente se definen con recursión recuerda en mucho a la de listas. Y las listas son una estructura que se deja manipular muy fácilmente por la recursión.

Por ejemplo, para calcular cuánto mide una lista, necesitaríamos algo así

```
int longitud (Lista lista) {

    if (lista == null) {
        return 0;
    }

    return 1+longitud (lista.siguiente());
}
```

Cuando usemos listas con recursión, generalmente la cláusula de escape será comprobar si la lista es nula.

Hasta ahora, habíamos utilizado iteraciones para tratar con las listas. A partir de ahora, trataremos de usar recursión siempre que sea posible, ya que es la forma natural de tratarlas dada su definición. Además, hará más legible nuestro código y nos dará más herramientas al momento de atacar problemas.

Sin embargo, deberá quedar claro que las listas pueden tratarse recursiva o iterativamente.

8.3.4. Arreglos y recursión

Así como las listas se manejan naturalmente con recursión, los arreglos se manejan naturalmente con iteración. Y sin embargo, así como las listas pueden ser manejadas de las dos formas, los arreglos también

```
void imprimeArreglo (Consola c, int[] miArreglo, int i) {

    if (i >= miArreglo.length) {
       return;
    }

    c.imprimeln ("Elemento␣"+i+":␣"+miArreglo[i]);
    imprimeArreglo (c, miArreglo, ++i);
}
```

De hecho, habrá ocasiones en que tendrá más sentido que en el ejercicio de arriba.

8.4. Ejercicios

1. Cambia las funciones agregaRegistro y quitaRegistro de la clase BaseDeDatos para que funcionen recursivamente.

   Debes probar que los métodos funcionen igual que antes en la clase de uso. Es posible que necesites una función auxiliar (privada seguramente) para que ésa sea sobre la que realmente se haga la recursión.

   La recursión puede ser engañosa; presta atención a los parámetros y a los valores de regreso. Y sobre todo *no pierdas nunca de vista la cláusula de escape.*

2. Para que compruebes que tu método quitaRegistro funciona correctamente, en tu clase de uso busca algún registro, y si existe, bórralo, y después vuélvelo a buscar.

Date cuenta que de nuevo no debemos cambiar los comentarios de JavaDoc; las funciones siguen haciendo lo mismo, aunque ya no lo hacen de la misma forma. Por esto documentamos *qué* hace una función, no *cómo*.

8.5. Preguntas

1. ¿Cómo te parece mejor que funcionan los métodos de la clase BaseDeDatos, iterativa o recursivamente? Justifica.
2. Si te das cuenta, hemos hasta ahora detenido el diseño de la Base de Datos, ya sólo hemos cambiado cómo se hacen por dentro las cosas. ¿Cómo crees que podríamos mejorar el diseño de la Base de Datos?

Manejo de Excepciones

If a program is useful, it will have to be changed.

– Murphy's Laws of Computer Programming #9

Semana:	Duodécima semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Excepciones.
Objetivo:	Aprender a manejar y crear excepciones.

Esta práctica se realiza durante la decimosegunda semana del curso. El profesor debe haber comenzado ya con excepciones, y hacer con ellas tantos ejemplos como sea posible.

Las excepciones son de las características más poderosas de Java. Fomentan la creación de programas más robustos e inteligentes, y facilita la comunicación con el usuario al ser los programas más inteligentes para detectar errores.

Lamentablemente, asumen demasiados prerrequisitos como para verlas desde el principio en un primer curso de programación. Para entender el funcionamiento de las excepciones hay que entender herencia y comprender cómo funciona la pila de ejecución.

La práctica es relativamente corta y sencilla. En esta práctica se les permite a los alumnos ver el código de dos de los paquetes que se les han proporcionado como *jarfiles*, porque ya deberían poder entender su funcionamiento. Los paquetes icc1_1.interfaz y icc1_2.interfaz se dejan para la próxima práctica, donde se verán interfaces gráficas.

Práctica 9
Manejo de Excepciones

9.1. Meta

Que el alumno aprenda a manejar y crear excepciones.

9.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- entender qué son las excepciones,
- manejar excepciones, y
- crear sus propias excepciones.

9.3. Desarrollo

Cuando en nuestra Base de Datos buscamos un registro y éste no existe, regresamos **null**. Ésta es una práctica muy común en computación: regresar un valor especial en caso de error o de que algo extraño haya ocurrido. Valores como **null**, -1, o **false** son muy usados para reportar una situación extraordinaria.

Empero, al complicar más un programa, el número de errores que podemos detectar va creciendo, y estar asignando un valor distinto de retorno a cada tipo de error puede resultar complicado (tenemos un montón de números negativos, pero sólo un **null**).

Además, si estamos dentro de una función recursiva, es posible que tengamos que escapar de la recursión tan rápido como sea posible, y eso muchas veces significa saltarse de alguna manera la cláusula de escape, y descargar la pila de ejecución del método.

Para todo esto surgió la idea de las excepciones.

9.3.1. Uso de excepciones

La idea detrás de las excepciones es justamente eso: situaciones de excepción en las que el programa debe de alguna manera detenerse a pensar qué está ocurriendo, ver si puede manejar la situación, o morir graciosamente.

Un programa nunca debe explotarle en la cara a un usuario. Si un error ocurre debe entonces tratar de encontrar la manera de continuar la ejecución del programa, o terminar con un mensaje de error claro y conciso.

La idea es sencilla: cuando haya métodos que se prestan a situaciones de excepción, tendrán la capacidad de *lanzar* excepciones (*throw* en inglés).

Si existe la posibilidad de que un método usado por el programa lance una o más excepciones, primero deberá *intentarse* (*try* en inglés) ejecutar el método. Si la situación extraordinaria ocurre en el intento, la función lanzará una excepción (sólo una), que debe ser *atrapada* (*catch* en inglés).

Por último (*finally*), habrá cosas que hacer después de ejecutar la función, independientemente de si algo malo ocurrió en el intento.

Todo esto se logra con los bloques **try** ... **catch** ... **finally**.

try ... catch ... finally

La sintaxis general para usar funciones que lancen excepciones es

```
try {
    // Invocación a métodos potencialmente peligrosas.
} catch (AlgunaExcepcion e1) {
    // Manejo de error (exception-handler).
} catch (OtraExcepcion e2) {
    // Manejo de error (exception-handler).
} catch (UnaExcepcionMas e3) {
    // Manejo de error (exception-handler).
...
} catch (UnaUltimaExcepcion eN) {
    // Manejo de error (exception-handler).
} finally {
    // Limpieza.
}
```

Dentro del bloque **try** se ejecutan funciones *peligrosas*. Con peligrosas queremos decir que mientras la función se ejecute pueden ocurrir situaciones de excepción, que dentro de la misma función no hay manera de manejar.[1]

En estas prácticas ya hemos visto clases en las que dentro de las funciones pueden ocurrir muchísimas excepciones; las clases del paquete icc1_1.es. Cuando hacemos entrada y salida, hay muchos errores potenciales, como por ejemplo:

- que no tengamos permisos para abrir un archivo,
- que no podamos conectarnos a una máquina del otro lado del mundo,
- que el espacio en disco se haya acabado,
- que se vaya la luz,

[1]Siempre hay manera de *manejar* errores; una de ellas es terminar el programa con un mensaje de error (que es lo que hacen todas las bibliotecas del curso hasta ahora). Lo que se quiere decir es que no hay manera general *correcta* de manejarlas.

- que desconecten el módem,
- que el sistema operativo se trabe,
- que la computadora explote en llamas,
- que tiemble la tierra,
- que el fin del mundo nos alcance, ...

Podemos seguir con muchas otras, pero lo importante es que al efectuar entrada y salida no podemos asegurar ninguna de las operaciones; abrir, escribir/leer, cerrar.

Dentro del bloque **try** envolvemos entonces a las posibles situaciones de excepción

```
try {
    EntradaDeDisco edd = null;
    edd = new EntradaDeDisco ("datos.dat"); // Abrimos el archivo.
    nombre    = edd.leeString ();           // Leemos datos.
    direccion = edd.leeString ();
    telefono  = edd.leeInteger ();
    edd.cierra ();                          // Cerramos el archivo.
}
```

Por supuesto, dentro del bloque **try** puede haber instrucciones inofensivas; la idea es agrupar a un conjunto de instrucciones peligrosas en un solo bloque **try** (aunque se mezclen instrucciones inofensivas), para no tener que utilizar un **try** para cada instrucción peligrosa.

En el momento en que una de las funciones lance una excepción, la ejecución del **try** se detendrá y se pasará a los bloques **catch**, también llamados *manejadores de excepción*. Es importante entender que la ejecución se detendrá *en el momento en que la excepción sea lanzada*. Esto quiere decir que cuando ejecutemos las líneas

```
EntradaDeDisco edd = null;
edd = new EntradaDeDisco ("datos.dat"); // Abrimos el archivo.
```

si el constructor de EntradaDeDisco lanza una excepción, entonces edd continuará valiendo **null** en el bloque **catch**, ya que nunca se realizó la asignación; la excepción fue lanzada antes.

Si la excepción es lanzada, la ejecución pasa entonces a los manejadores de excepción, los **catch**, donde es atrapada por el manejador que le corresponda.

```
catch (ExcepcionArchivoInexistente eai) {
    c.println ("El␣archivo␣\"datos.dat\"␣no␣existe.");
} catch (ExcepcionDeEntrada ede) {
    c.println ("No␣se␣pudo␣leer␣el␣archivo.");
}
```

Por último, la ejecución pasa al bloque **finally** (si existe), que se ejecuta haya o no habido una excepción. Es para realizar operaciones críticas, sin importar si hubo o no errores en el bloque **try**, o el tipo de estos errores, si es que hubo.

```
    finally {
        terminaTareas ();
    }
```

El código que hemos descrito quedaría de la siguiente forma

```
    try {
        EntradaDeDisco edd = null;
        edd = new EntradaDeDisco ("datos.dat"); // Abrimos el archivo.
        nombre    = edd.leeString ();           // Leemos datos.
        direccion = edd.leeString ();
        telefono  = edd.leeInt ();
        edd.cierra ();                          // Cerramos el archivo.
    } catch (ExcepcionArchivoInexistente eai) {
        c.println ("El archivo \"datos.dat\" no existe.");
    } catch (ExceptionDeEntrada ede) {
        c.println ("No se pudo leer el archivo.");
    } finally {
        terminaTareas ();
    }
```

En este ejemplo hay que notar que dentro del **try** se llevan a cabo todas las operaciones relacionadas con leer de disco.

El bloque **finally** es opcional; no es obligatorio que aparezca. Pero debe quedar en claro que cuando aparece, el bloque **finally** se ejecuta *incondicionalmente*, se lance o no se lance ninguna excepción. La única manera de impedir que un **finally** se ejecute es terminar la ejecución del programa dentro del **catch** (lo que no tiene mucho sentido si existe un bloque **finally**).

La parte más importante al momento de manejar excepciones es, justamente, los manejadores de excepciones. Es ahí donde determinamos qué hacer con la excepción.

Para utilizar los manejadores de excepciones, necesitamos utilizar a las clases herederas de Throwable.

Actividad 9.1 Con el apoyo de tu ayudante, consulta la documentación de la clase Throwable.

La clase Throwable

La sintaxis de un manejador de excepción es

```
    catch (<AlgunaClaseHerederaDeThrowable> t) {
        ...
    }
```

Donde <AlgunaClaseHerederaDeThrowable> es alguna clase heredera de la clase Throwable (*lanzable*, en inglés). En particular, las clases Exception y Error son herederas de Throwable. El

compilador de Java protesta si en un manejador de excepción tratamos de utilizar una clase que no sea heredera en algún grado de la clase Throwable.

Las excepciones son objetos comunes y corrientes de Java, con métodos y variables que pueden utilizarse para obtener más información de la situación extraordinaria que ocurrió.

Por ejemplo, todos los objetos de clases herederas de Throwable tienen la función printStackTrace, que imprime información acerca de en qué método ocurrió el error y qué otros métodos fueron llamados antes que él (lo que se conoce como la *pila de ejecución*).

Si una función es capaz de lanzar n tipos de excepciones (o sea, n distintas clases herederas de Throwable), entonces debemos atrapar las n excepciones; el compilador no permitirá la compilación si no lo hacemos.

Si debemos atrapar distintas excepciones que son herederas de una clase, podemos sólo atrapar su superclase. En particular, el siguiente código es válido en la gran mayoría de los casos (ya que casi todas las excepciones heredan a Exception):

```
try {
    // Cosas peligrosas que lanzan chorrocientas excepciones
    // distintas.
} catch (Exception e) {
    c.imprimeln ("Algo␣malo␣ocurrió.").
}
```

Sin embargo es considerada una mala práctica, ya que perdemos todo el fino control que nos dan las excepciones (no distinguimos qué excepción estamos atrapando), y será prohibido su uso en estas prácticas. Empero, atrapar a la clase Exception estará permitido si es al final, cuando ya se ha escrito el manejador de todas las excepciones posibles que pueden lanzarse.

Lanzamiento de excepciones

Hay dos maneras de manejar las excepciones dentro de nuestros métodos. La primera es la que ya vimos, atrapar las excepciones en un bloque **try ... catch ... finally**. La segunda es *encadenar* el lanzamiento.

Encadenar el lanzamiento significa que asumimos que nuestro método no es el responsable de manejar la excepción, sino que debe ser responsabilidad de quien sea que llame a nuestro método. Entonces, nuestro método en lugar de atrapar la excepción volverá a lanzarla.

Piensen en la situación de excepción como en una papa caliente, y en los métodos como una fila de personas. La primera persona (o sea método) que de repente tenga la papa caliente en las manos, tiene la opción de manejar la papa (resolver el problema, manejar la excepción), o pasárselo a la persona que sigue, lavándose las manos. La papa (o excepción) puede entonces ir recorriendo personas (métodos) en la fila (en la pila de ejecución) hasta que algún método maneje la excepción en lugar de lanzarla de nuevo.

Para lanzar de nuevo la excepción, tenemos que hacer específico que nuestro método puede lanzar esa excepción utilizando la cláusula **throws** en el encabezado, por ejemplo:

```
public void miFuncionLanzadora (int arg) throws AlgunaExcepcion {
```

Una vez que nuestro método miFuncionLanzadora declara que puede lanzar a la excepción AlgunaExcepcion, ya no es necesario que envolvamos con un **try** a ninguna función que pudiera

lanzar también a la excepción AlgunaExcepcion. El lanzamiento queda encadenado automáticamente.

Por supuesto, podemos manejar distintos tipos de papas calientes; nada más debemos especificarlo en el encabezado

```
public void miFuncionLanzadora (int arg)
    throws AlgunaExcepcion, OtraExcepcion, UnaExcepcionMas {
```

Un método puede ser capaz de lanzar un número arbitrario de excepciones, y cuando sea llamado el método deberá ser dentro de un bloque **try** (a menos que la función que llame al método también lance las mismas excepciones).

Si llamamos a un método que puede lanzar excepciones, y el método desde donde lo llamamos no lanza las mismas excepciones, entonces hay que envolver la llamada con un **try**; de otra manera, el compilador protestará.

El método **main** puede lanzar excepciones, que son atrapadas por la JVM. Es una forma de probar métodos que lanzan excepciones sin necesitar escribir los bloques **try** ... **catch** ... **finally** (sencillamente encadenamos **main** al lanzamiento de las excepciones).

9.3.2. Creación de excepciones

Java provee muchísimas excepciones, destinadas a muchos tipos de problemas que pueden ocurrir en un programa, pero en cuanto un programa comienza a crecer en complejidad ciertos problemas inherentes al programa aparecen.

Por ejemplo, mucha gente puede creer que en una Base de Datos si se trata de agregar un registro idéntico a uno ya existente, es una situación lo suficientemente singular como para lanzar una excepción.

Entonces quisiéramos crear excepciones por nuestra cuenta. Para hacerlo, sólo tenemos que crear una clase que herede a Exception[2]

```
public class RegistroDuplicado extends Exception {}
```

Noten que la definición de la clase está completa. Generalmente no habrá necesidad de crear funciones especiales para las excepciones; sólo las utilizaremos para distinguir los errores de nuestros programas, y para ello nos bastarán los métodos heredados de la clase Exception.

Esto no quiere decir que no podamos sobrecargar las funciones de la clase Exception, ni que no sea necesario nunca. Simplemente, en la mayoría de los casos podremos ahorrárnoslo.

Lo que sí muchas veces haremos será crear una jerarquía de clases exclusivamente para excepciones de nuestro programa. Esto facilitará muchas cosas al momento de definir qué excepciones puede lanzar una clase, o al momento de escribir los bloques **try** ... **catch** ... **finally**.

Si queremos que una función lance una excepción creada por nosotros, al igual que cuando lanzaba excepciones creadas por alguien más, necesitamos que en el encabezado especifique que puede lanzar la excepción, y después necesitamos crear la excepción con **new**, para poder lanzarla con **throw**.

[2] O cualquier heredero de la clase Throwable, incluso podemos heredar a Throwable directamente, aunque generalmente será a Exception a la que extendamos.

```
public void miFuncionLanzadora (int arg)
    throws MiExcepcion1, MiExcepcion2 {
    ...
    if (algoMalo) {
        throw new MiExcepcion1 ();
    }
    ...
    if (algoPeor) {
        throw new MiExcepcion2 ();
    }
    ...
}
```

Si las clases MiExcepcion1 y MiExcepcion2 son herederas de la clase MiSuperExcepcion, podemos reducir el código de arriba como sigue:

```
public void miFuncionLanzadora (int arg) throws MiSuperExcepcion {
    ...
    if (algoMalo) {
        throw new MiExcepcion1 ();
    }
    ...
    if (algoPeor) {
        throw new MiExcepcion2 ();
    }
    ...
}
```

Se darán cuenta de que lanzar las excepciones es algo que se decide en tiempo de ejecución (por eso los ifs). No tiene sentido hacer una función que *siempre* lance una excepción. Las excepciones son así, singulares, y deben ser lanzadas después de hacer una evaluación acerca del estado del programa (como ver que un nuevo registro es idéntico a uno ya existente).

9.3.3. El flujo de ejecución

Al momento de hacer un **throw**, la ejecución de la función termina, y la excepción recorre todas las funciones que han sido llamadas, deteniéndolas, hasta que encuentra un bloque **catch** que lo atrape. Eso rompe cualquier algoritmo recursivo, cualquier iteración y en general cualquier flujo de ejecución, lo que hace al **throw** mucho más poderoso que el **return**.

Dado que se pueden descartar muchas llamadas a funciones que se hayan hecho (desmontándolas en la pila de ejecución), una excepción en general debe tratar de *manejarse* más que de *repararse*. Repararla requeriría que se volvieran a hacer las llamadas detenidas por el lanzamiento de la excepción, con el posible riesgo de que el error ocurra de nuevo.

Lo sensato es dar un mensaje al usuario de que lo que pidió no puede realizarse, y regresar a un punto de ejecución seguro en el programa (el menú inicial o la pantalla de inicio).

9.3.4. La clase RuntimeException

Al inicio de la práctica se mencionó, un poco en tono de broma, lo que puede ir mal en el momento de ejecutar un programa.

Aunque ciertamente si nos llega el fin del mundo lo que menos nos va a importar es si nuestro programa funciona o no, también es cierto que hay cosas que ocurren que no tienen que ver con que nuestro programa esté bien o mal escrito o con que el usuario pidiera algo con sentido o no.

La memoria y el disco se acaban, o fallan, las conexiones se cortan, la luz se va. Pero existen cosas todavía más sencillas que también pueden ocurrir y que harán abortar al programa: una división por cero, tratar de utilizar una referencia nula, salirnos del rango de un arreglo.

Todos estos errores en Java también son excepciones, pero no hay necesidad de atraparlas en bloques **try** ... **catch** ... **finally**. Y no hay necesidad de hacerlo porque estos errores ocurren no en el contexto de llamar un método, sino en el contexto de un simple instrucción.

Todas estas excepciones son herederas de la mal llamada clase RuntimeException[3], que a su vez es heredera de Exception. Las excepciones herederas de la clase RuntimeException no son situaciones excepcionales, son descuidos del programador.

Si alguien no quiere que una excepción creada por él tenga necesidad de ser atrapada, sencillamente tiene que hacerla heredera de RuntimeException y el compilador dejará que el método sea llamado aun cuando no se trate de atrapar ninguna excepción. Tampoco es necesario que se especifiquen en el encabezado con la cláusula **throws**. Pero se considera un error hacer las excepciones de un programa herederas de la clase RuntimeException, ya que como se dijo no son situaciones excepcionales, sino descuidos del programador.

En partes críticas de código, puede atraparse una excepción de la clase RuntimeException al final de los manejadores de excepciones para asegurar que nada malo ocurra. Sin embargo, casi todas las excepciones herederas de la clase RuntimeException pueden prevenirse (comprobar que el denominador no sea cero, comprobar que la referencia no sea nula, siempre revisar los límites de un arreglo), así que es mejor programar cuidadosamente.

Sin embargo, mientras se está depurando un programa puede ser muy útil atrapar las excepciones de la clase RuntimeException, sobre todo en los casos en que no parece haber una razón por la que el programa truena.

9.4. Ejercicios

Hasta ahora, las excepciones de nuestros *jarfiles* habían sido suprimidas. Para que se den una idea, las excepciones se manejaban así

```
try {
    r = Integer.parseInt (s);
} catch (NumberFormatException nfe) {
    System.err.println ("***␣ERROR␣***:␣\""+s+
                        "\"␣no␣es␣un␣byte␣válido.");
    System.err.println ("***␣ERROR␣***:␣Ejecución␣"+
                        "del␣programa␣detenida.");
    System.exit (1);
}
```

[3]Mal llamada porque *todas* las excepciones ocurren en tiempo de ejecución.

El método exit de la clase System (como ya habrán adivinado) obliga a que un programa termine.

El manejo de excepciones se suprimió para que no tuvieran la necesidad de preocuparse por ellas y se pudieran concentrar en resolver los problemas que se les planteaban.

Mas las excepciones son de las características que hacen de Java un buen lenguaje de programación, y su uso debe ser fomentado y diseminado.

A partir de ahora, todas las bibliotecas usadas serán reemplazadas por versiones que utilizan excepciones en sus métodos, excepciones que deberás atrapar y manejar de manera correcta en tus programas.

Baja las nuevas versiones de los paquetes que se han utilizado. Los nuevos *jarfiles* son `interfaz2.jar`, `util2.jar` y `es2.jar`. También debes consultar la documentación de cada uno de los paquetes para ver qué excepciones lanzan los métodos que hemos usado hasta ahora.

Los paquetes ya no se llaman icc1_1.interfaz, icc1_1.util ni icc1_1.es. Ahora tendrán el prefijo icc1_2, por lo que sus nuevos nombres son icc1_2.interfaz, icc1_2.util, icc1_2.es. El código fuente de los paquetes icc1_2.util y icc1_2.es también estará disponible por si quieres ver cómo están implementadas y tal vez usar un poco del código dentro de ellas. También estará disponible el código de los paquetes icc1_1.util y icc1_1.es, para que veas cómo manejaban las excepciones.

Nota que las clases Lista y (aunque ya no la usamos) ListaDeCadena solamente lanzan la excepción NullPointerException, que es heredera de la clase RuntimeException. Por tanto, no es necesario que utilices un try para manejar listas (sólo programa cuidadosamente).

En el caso particular del paquete icc1_1.es, se había hecho justamente para que no tuvieran que preocuparse por las excepciones cuando quisieran trabajar con el disco duro. Ahora que van a tener que manejar excepciones, no tiene realmente mucho sentido que sigan usando el paquete icc1_2.es.

Si quieren, pueden comenzar a utilizar las clases del paquete java.io; pero si quieren pueden seguir usando al paquete icc1_2.es. Lo que sí está prohibido es usar cualquier paquete icc1_1.*.

1. Haz que tu Base de Datos (tal y como la dejaste la práctica pasada) compile con los nuevos paquetes. Asegúrate de cambiar los **import** para que ahora importen a los paquetes de icc1_2 en lugar de los de icc1_1.

   Asegúrate también de que en el directorio donde estés trabajando no quede ninguna copia de los antiguos *jarfiles*, y que la bandera `-classpath` de `javac` y de `java` utilice los nuevos *jarfiles*.

   Vas a tener que utilizar bloques **try** ... **catch** ... **finally** para manejar las excepciones que lanzan los nuevos paquetes. Puedes hacer lo que creas conveniente en los manejadores de excepción, pero trata de hacer algo. No siempre se puede; en particular, si algo sale mal al leer o escribir en disco, realmente no hay mucho que puedas hacer. Eso sí, continúa la ejecución del programa (no uses el método exit de la clase System).

   Recuerda: las excepciones son clases. Para utilizar las excepciones de un paquete, debes importar la excepción al inicio de la clase. Por ejemplo, para atrapar la excepción ArchivoNoEncontrado, del paquete icc1_2.es, debes poner al inicio de tu clase

```
import icc1_2.es.ArchivoNoEncontrado;
```

2. En la clase BaseDeDatos, modifica las funciones agregaRegistro y borraRegistro para que lancen excepciones (distintas) en los siguientes casos

   - cuando se trate de agregar un registro duplicado,
   - cuando se trate de borrar un registro que no existe,

   Tienes que crear las clases de las excepciones. Llámalas como tú quieras y utiliza la jerarquía de clases que consideres necesaria, pero que las excepciones estén jerarquizadas de manera no trivial (o sea, que no todas sean herederas directas de la clase Exception).

   Una excepción muy usada en la biblioteca de clases de Java es la excepción NullPointerException, que es lanzada cada vez que se trata de usar una referencia nula (**null**) en un contexto en el que está prohibido. Esta excepción es heredera de la clase RuntimeException, así que no es obligatorio atraparla.

   Haz que tus métodos lancen también la excepción NullPointerException cuando el parámetro que reciban sea **null**. No es necesario usar ningún **import** para usar la clase NullPointerException.

   Una vez modificadas las funciones, tendrás que modificar la clase de uso, ya que se necesitará atrapar las excepciones para que compile la clase de nuevo.

3. Cuando modifiques las funciones agregaRegistro y borraRegistro, modifica los comentarios de JavaDoc utilizando la etiqueta @throws para especificar qué excepciones lanza y cuándo las lanza.

   También comenta las clases de tus excepciones.

9.5. Preguntas

1. ¿Crees que es útil el manejo de excepciones? Justifica.
2. ¿Qué otras excepciones crees que podrían crearse para nuestra Base de Datos?

Interfaces Gráficas

If a program is useless, it will have to be documented.

– Murphy's Laws of Computer Programming #10

Semana:	Decimotercera semana.
Tiempo de entrega:	Dos semanas.
Prerrequisitos:	Eventos, interfaces gráficas.
Objetivo:	Aprender a manejar eventos y crear interfaces gráficas.

Esta práctica se realiza durante la decimotercera semana del curso. El profesor debe haber visto ya eventos y comenzar a ver interfaces gráficas. La práctica tiene 28 páginas, pero tenía que ser de esta manera porque había que explicar línea por línea un ejemplo con el uso de interfaces gráficas.

Las interfaces gráficas son sencillas en Java; sólo hay que tener bien clara la naturaleza asíncrona de los eventos y saber cómo puede desarrollarse la ejecución del programa.

Se les da dos semanas no porque sea difícil en sí misma la práctica, sino porque el proceso de depuración cuando se usan interfaces gráficas es más delicado, y pueden necesitar más tiempo. Además hay que dar oportunidad, a los que puedan, de lucirse con sus interfaces.

El curso está prácticamente terminando al llegar a este tema. La idea es que tengan el tiempo de divertirse con las interfaces y con los ejemplos y ejercicios.

Al llegar a este punto, se abre el último paquete que se les había proporcionado, y el curso cumple con su objetivo de proporcionarle al estudiante herramientas para facilitar ciertos temas complejos, y hacer que el propio alumno pueda hacer esas herramientas.

No se les deja como tarea que hagan las cajas negras; mas se les ha proporcionado el material práctico y teórico suficiente para que ellos puedan comprender cómo funcionan.

Práctica 10
Interfaces Gráficas

10.1. Meta

Que el alumno aprenda a crear interfaces gráficas.

10.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- entender las interfaces gráficas en Java,
- entender qué son los eventos, y
- crear interfaces gráficas.

10.3. Desarrollo

La mayor parte del mundo utiliza la computadora sólo como una herramienta. Escriben trabajos en ellas, calculan sus impuestos, organizan su agenda, platican con alguien del otro lado del mundo, le mandan un correo electrónico al hijo que está estudiando lejos, etc. Para hacer todo eso necesitan aplicaciones sencillas de usar, que les permita realizar su trabajo y pasatiempos sin muchas complicaciones.

Cuando se dice que estas aplicaciones deben ser sencillas de usar, muchos coinciden en que la mejor manera de facilitarle la vida al usuario es que el programa se comunique con él a través de Interfaces Gráficas de Usuario, IGUs, o GUIs por las mismas siglas en inglés.

(El hecho de que las interfaces gráficas nos compliquen la vida a los programadores no parece importarle mucho a los usuarios...)

A lo largo de estas prácticas hemos utilizado interfaces gráficas sencillas, aunque ustedes mismos no las han programado. En esta práctica veremos cómo se programan las interfaces gráficas, y discutiremos la característica más importante de ellas, y que se ha evitado intencionalmente hasta ahora. Nos referimos al manejo de eventos.

Las primeras versiones de Java utilizaban las clases del paquete java.awt para crear interfaces gráficas. Las últimas versiones utilizan las clases del paquete javax.swing, que será el definitivo una vez que las aplicaciones escritas con AWT sean convertidas a Swing. En esta práctica utilizaremos Swing.

Actividad 10.1 Con el apoyo de tu ayudante, ve las clases del paquete javax.swing y java.awt.

10.3.1. Componentes

Una interfaz gráfica en Java está compuesta de componentes alojados en un contenedor. Veamos una pequeña interfaz gráfica que se ha usado a lo largo del curso, la pequeña ventana de diálogo que aparece cuando hacemos

```
c.leeString ("Mete una cadena:");
```

La ventana la puedes ver en la figura 10.1

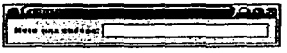


Figura 10.1: Diálogo de leeString

En esta ventana de diálogo hay cuatro componentes:

- Un componente de la clase JDialog, un *diálogo*. Éste es un contenedor de *primer nivel*. Este tipo de componentes es generalmente una ventana, y los más usuales son JFrame, JDialog y Applet.
- Un componente de la clase JPanel, un *panel*. Éste es un contenedor *intermedio*, y lo utilizaremos para agrupar de manera sencilla otros componentes.
- Un componente de la clase JLabel, una *etiqueta*. Las etiquetas generalmente serán sólo usadas para mostrar algún tipo de texto, aunque pueden mostrar imágenes (*pixmaps*).
- Un componente de la clase JTextField, un *campo de texto*. Son componentes usados casi siempre para obtener del usuario una cadena de pocos caracteres o de una sola línea.

Los componentes en una interfaz gráfica están ordenados jerárquicamente. En el primer nivel está el diálogo, dentro de él está el panel, y dentro del panel están la etiqueta y la caja de texto, como se muestra en la figura 10.2.

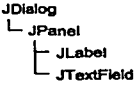


Figura 10.2: Jerarquía de componentes

El diálogo, que en este sencillo ejemplo es la ventana principal, sirve más que nada para poner juntos a los demás componentes. Es un *contenedor de primer nivel.*

El panel es un *contenedor intermedio.* Su propósito en la vida es ayudarnos a acomodar otros componentes.

La etiqueta y la caja de texto son *componentes atómicos*, componentes que no contienen a otros componentes, sino que por sí mismos muestran o reciben información del usuario.

En el pequeño diagrama de la jerarquía de componentes nos saltamos algunos componentes que hay entre el diálogo y el panel. Sin embargo, la mayor parte de las veces no habrá que preocuparse de esos componentes.

Para crear esa interfaz gráfica, el código fundamental es

```
JDialog     dialogo    = new JDialog (...);
JPanel      panel      = new JPanel (...);
JLabel      etiqueta   = new JLabel (...);
JTextField  cajaTexto  = new JTextField (...);

panel.add (etiqueta);
panel.add (cajaTexto);

dialog.getContentPane().add (panel);

dialog.pack ();
dialog.setVisible (true);
```

10.3.2. Administración de trazado

Los componentes se dibujan sobre un contenedor intermedio (como son los objetos de la clase JPanel) utilizando un *administrador de trazado* (*layout manager*).

La administración de trazado es de las características más cómodas de Swing, ya que prácticamente sólo hay que especificar cómo queremos que se acomoden los componentes, y Swing se encarga de calcular los tamaños y posiciones para que los componentes se vean bien distribuidos y con un tamaño agradable.

Sin embargo puede volverse un poco complicado el asunto, ya que hay muchos administradores de trazado, y de acuerdo a cuál se escoja, nuestra aplicación terminará viéndose muy distinta.

A lo largo de la práctica veremos distintos ejemplos de los administradores de trazado más comunes.

10.3.3. Eventos

Ya sabemos cómo se ponen algunos componentes en una pequeña ventana. Ahora, ¿cómo hacemos para que la interfaz *reaccione* a lo que el usuario haga? Por ejemplo, la caja de texto de nuestro diálogo debe cerrar el diálogo cuando el usuario presione la tecla ENTER en ella.

Esto se logra con el manejo de *eventos.* Un evento ocurre cuando el usuario genera algún cambio en el estado de un componente: teclear en una caja de texto, hacer click con el ratón en un botón, cerrar una ventana.

Si existe un *escucha* (*listener* en inglés) para el objeto que emitió el evento, entonces podrá ejecutar su *manejador del evento*. Además, un objeto puede tener más de un escucha, cada uno de los cuales ejecutará su manejador del evento si el evento ocurre.

Entender cómo se emiten los eventos y cómo se manejan es lo más complicado de hacer interfaces gráficas. Lo demás es seguir algunas recetas para utilizar los distintos tipos de componentes, que son muchos, pero que todos funcionan de manera similar.

Escuchas

Los escuchas son interfaces; son como clases vacías que nos dicen qué tipos de eventos se pueden emitir, pero que nos dejan la libertad de implementar qué hacer cuando los eventos sucedan.

Hagamos un ejemplo completo: una pequeña ventana con una etiqueta para mensajes y un botón. Primero hagámoslo sin escuchas

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;
3  import javax.swing.JLabel;
4  import javax.swing.JPanel;
5  import javax.swing.BorderFactory;
6  import java.awt.BorderLayout;
7  import java.awt.GridLayout;
8  import java.awt.Container;
9
10 public class Ejemplo1 {
11     public static void main (String[] args) {
12         JFrame frame = new JFrame ("Ejemplo 1");
13         JButton boton = new JButton ("Haz click");
14         JLabel etiqueta = new JLabel ("Esto es una etiqueta");
15
16         JPanel panel = new JPanel ();
17         panel.setBorder (BorderFactory.createEmptyBorder (5,5,5,5));
18         panel.setLayout (new GridLayout (0, 1));
19
20         panel.add (boton);
21         panel.add (etiqueta);
22
23         Container c = frame.getContentPane ();
24         c.add (panel, BorderLayout.CENTER);
25
26         frame.pack ();
27         frame.setVisible (true);
28     }
29 }
```

¿Qué hace el programa? En las líneas 12, 13 y 14 crea los tres principales componentes de nuestra aplicación: nuestra ventana principal (JFrame), nuestro botón (JButton), y nuestra etiqueta (JLabel). Esos son los componentes que el usuario verá directamente.

Lo siguiente que hace el programa es crear un panel (línea 16). El único propósito del panel es agrupar a nuestro botón y a nuestra etiqueta en la ventana principal. Lo primero que hace es ponerse un borde de 5 pixeles alrededor [17], y definirse un administrador de trazado [18]; en este caso es de la clase GridLayout, lo que significa que será una cuadrícula. Ya que le pasamos como parámetros 0 y 1, se entiende que tendrá un número no determinado de renglones (de ahí el 0), y una sola columna. Lo último que hace el panel es añadirse el botón y la etiqueta [20,21].

El programa después obtiene la ventana contenedora de la ventana principal [23] (todos los contenedores de primer nivel tienen una ventana contenedora), y a áquel le añadimos el panel [24]. Para terminar, la ventana principal se *empaca* y se hace visible. Durante el proceso de empacado se calculan los tamaños de los componentes, para que automáticamente se defina el tamaño de la ventana principal.

Actividad 10.2 Escribe la clase `Ejemplo1.java`, compílala y corre el ejemplo.

El programa no hace nada interesante (ni siquiera termina cuando se cierra la ventana). El botón puede ser presionado cuantas veces queramos; pero nada ocurre porque no le añadimos ningún escucha. Cuando el usuario hace click en el botón se dispara un evento; pero no hay ningún manejador de evento que se encargue de él.

Vamos a escribir un escucha para nuestro ejemplo. Ya que el evento que nos interesa es el click del botón, utilizaremos un *escucha de ratón*, o *mouse listener*, que es una interfaz que está declarada como sigue

```
1  public interface MouseListener extends EventListener {
2      public void mouseClicked (MouseEvent e);
3      public void mouseEntered (MouseEvent e);
4      public void mouseExited (MouseEvent e);
5      public void mousePressed (MouseEvent e);
6      public void mouseReleased (MouseEvent e);
7  }
```

(Aunque se pueden escribir escuchas propios, Java provee ya los escuchas más utilizados para distintos eventos, por lo que no hay necesidad de hacerlos.)

Esto nos dice que los escuchas de ratón ofrecen métodos para cuando el ratón haga click sobre el componente (mouseClicked), para cuando el ratón se posa sobre el componente (mouseEntered), para cuando el ratón deja de estar sobre el componente (mouseExited), para cuando se presiona el botón del ratón sin soltarlo (mousePressed), y para cuando se suelta el botón del ratón después de presionarlo (mouseRelease).

Si queremos hacer un escucha para nuestro botón, debemos crear una clase que implemente la interfaz MouseListener, diciendo qué debe hacerse en cada caso:

```
1  import java.awt.event.MouseListener;
2  import java.awt.event.MouseEvent;
3  import javax.swing.JLabel;
4
```

```
5  public class MiEscuchaDeRaton implements MouseListener {
6
7      private int     contador;
8      private JLabel  etiqueta;
9
10     public MiEscuchaDeRaton (JLabel etiqueta) {
11         this.etiqueta = etiqueta;
12         contador = 0;
13     }
14
15     public void mouseClicked (MouseEvent e) {
16         contador++;
17         etiqueta.setText ("Clicks:␣"+contador+".");
18     }
19
20     public void mouseEntered (MouseEvent e) {}
21     public void mouseExited (MouseEvent e) {}
22     public void mousePressed (MouseEvent e) {}
23     public void mouseReleased (MouseEvent e) {}
24 }
```

Realmente sólo implementamos la función mouseClicked; las otras no nos interesan. Pero debemos darles una implementación a todas (aunque sea como en este caso una implementación vacía), porque si no el compilador se quejará de que no estamos implementando todas las funciones de la interfaz MouseListener.

¿Qué hace este manejador de evento? Sólo se hace cargo del evento mouseClicked; cuando el usuario haga click sobre el botón, la etiqueta cambiará su mensaje a "Clicks" y el número de clicks que se hayan hecho.

Para que nuestro botón utilice este escucha sólo tenemos que agregarlo (modificando la clase Ejemplo1)

```
13 JButton boton = new JButton ("Haz␣click");
14 JLabel etiqueta = new JLabel ("Esto␣es␣una␣etiqueta");
15
16 MiEscuchaDeRaton escucha = new MiEscuchaDeRaton (etiqueta);
17 boton.addMouseListener (escucha);
```

Actividad 10.3 Escribe la clase `MiEscuchaDeRaton.java` y modifica `Ejemplo1.java` para que el botón utilice el escucha. Compila y ejecuta el programa de nuevo.

Adaptadores

Como vimos arriba, muchas veces no queremos implementar *todas* las funciones que ofrece un escucha. La mayor parte de las veces sólo nos interesa el evento de click sobre un botón, y no nos importará el resto.

Para esto están los *adaptadores*. Los adaptadores son clases concretas que implementan a las interfaces de un escucha, pero que no hacen nada en sus funciones; de esta forma, uno sólo hereda al adaptador y sobrecarga la función que le interese. Por ejemplo, el adaptador de ratón (*mouse adapter*) es

```
1  public class MouseAdapter implements MouseListener {
2      public void mouseClicked (MouseEvent e) {}
3      public void mouseEntered (MouseEvent e) {}
4      public void mouseExited (MouseEvent e) {}
5      public void mousePressed (MouseEvent e) {}
6      public void mouseReleased (MouseEvent e) {}
7  }
```

No hacen nada sus funciones; pero si únicamente nos interesa el evento del click del ratón sólo hay que implementar mouseClicked. Nuestra clase MiEscuchaDeRaton se reduciría a

```
1  import java.awt.event.MouseAdapter;
2  import java.awt.event.MouseEvent;
3  import javax.swing.JLabel;
4
5  public class MiEscuchaDeRaton extends MouseAdapter {
6
7      private int    contador;
8      private JLabel etiqueta;
9
10     public MiEscuchaDeRaton (JLabel etiqueta) {
11         this.etiqueta = etiqueta;
12         contador = 0;
13     }
14
15     public void mouseClicked (MouseEvent e) {
16         contador++;
17         etiqueta.setText ("Clicks:␣"+contador+".");
18     }
19 }
```

Como los otros eventos no nos interesan, los ignoramos. No hacemos nada si ocurren. Todos los escuchas de Java tienen un adaptador disponible para facilitarnos la vida.

Actividad 10.4 Vuelve a modificar `MiEscuchaDeRaton.java` para que extienda el adaptador de ratón. Compila y ejecuta de nuevo el programa.

Clases internas y anónimas

En estas prácticas no hemos mencionado que Java puede tener clases internas; clases declaradas dentro de otras clases

```
public class A {
    ...
    class B {
        ...
    }
}
```

La clase B es clase interna de A; sólo puede ser vista dentro de A. Esto es muy útil con los escuchas y adaptadores, ya que nada más nos interesa que los vean dentro de la clase donde estamos creando nuestra interfaz gráfica.

Cuando compilemos el archivo `A.java`, que es donde viven las clases A y B, se generarán dos archivos: `A.class`, y `A$B.class`; este último quiere decir que B es clase interna de A.

Podemos entonces crear nuestros escuchas y adaptadores dentro de la clase donde creemos nuestra interfaz gráfica. Pero aún podemos hacer más: podemos utilizar clases anónimas.

Las clases anónimas son clases creadas *al vuelo*, que no tienen nombre. Por ejemplo, para nuestro botón podríamos añadirle nuestro escucha de ratón al vuelo, de esta manera

```
14  final JLabel etiqueta = new JLabel ("Esto es una etiqueta");
15  final Integer contador = new Integer (0);
16
17  boton.addMouseListener (new MouseAdapter () {
18          public void mouseClicked (MouseEvent e) {
19              contador = new Integer (contador.intValue()+1);
20              etiqueta.setText ("Clicks: "+contador.intValue()+".");
21          }
22      });
```

Nuestra clase `Ejemplo1.java` quedaría así

```
1   import javax.swing.JFrame;
2   import javax.swing.JButton;
3   import javax.swing.JLabel;
4   import javax.swing.JPanel;
5   import javax.swing.BorderFactory;
6   import java.awt.BorderLayout;
7   import java.awt.GridLayout;
8   import java.awt.Container;
9   import java.awt.event.MouseAdapter;
10  import java.awt.event.MouseEvent;
11
12  public class Ejemplo1 {
13      public static void main (String[] args) {
14          JFrame frame = new JFrame ("Ejemplo 1");
15          JButton boton = new JButton ("Haz click");
16          final JLabel etiqueta = new JLabel ("Esto es una etiqueta");
17          final Integer contador = new Integer (0);
18
19          boton.addMouseListener (new MouseAdapter () {
```

```
20                      public void mouseClicked (MouseEvent e) {
21                          contador = new Integer (contador.intValue()+1);
22                          etiqueta.setText ("Clicks:␣"+contador.intValue()+".");
23                      }
24                  });
25
26              JPanel panel = new JPanel ();
27              panel.setBorder (BorderFactory.createEmptyBorder (5,5,5,5));
28              panel.setLayout (new GridLayout (0, 1));
29
30              panel.add (boton);
31              panel.add (etiqueta);
32
33              Container c = frame.getContentPane ();
34              c.add (panel, BorderLayout.CENTER);
35
36              frame.pack ();
37              frame.setVisible (true);
38          }
39      }
```

Actividad 10.5 Vuelve a modificar el archivo `Ejemplo1.java` para que utilice la clase anónima. Compílalo y ejecútalo de nuevo.

Ahí estamos creando una clase anónima al vuelo al invocar la función addMouseListener. Al compilar `Ejemplo1.java`, la clase anónima se compilará en el archivo `Ejemplo1$1.class`.

Las clases anónimas son mucho más cómodas de utilizar que el tener que crear una clase para cada uno de los eventos de un programa (el número de eventos puede aumentar a cientos de ellos).

Muchos notarán (y se preguntarán), por qué ahora la etiqueta y el contador son variables finales, y por qué el contador es un objeto de la clase Integer. La mayor desventaja de las clases anónimas es que no pueden utilizar variables locales declaradas fuera de ellas mismas a menos que sean finales. En el caso de la etiqueta realmente no importa ya que no modificamos la referencia al objeto (cambiamos su contenido, pero la referencia sigue siendo la misma). En el caso del contador queremos que se modifique en cada llamada, y entonces no podemos usar una variable de tipo int, ya que tiene que ser final. Por eso usamos un objeto de la clase Integer.

Parte del problema en el ejemplo es que todo el código está en el método main; se puede resolver todo de manera más elegante si rediseñamos un poco:

```
1   import javax.swing.BorderFactory;
2   import javax.swing.JButton;
3   import javax.swing.JFrame;
4   import javax.swing.JLabel;
5   import javax.swing.JPanel;
6   import java.awt.BorderLayout;
```

```
7  import java.awt.Container;
8  import java.awt.GridLayout;
9  import java.awt.event.MouseAdapter;
10 import java.awt.event.MouseEvent;
11 import java.awt.event.WindowAdapter;
12 import java.awt.event.WindowEvent;
13
14 public class Ejemplo1 {
15
16     int contador;
17
18     public Ejemplo1 () {
19         contador = 0;
20     }
21
22     public JFrame dameVentana () {
23         JFrame frame = new JFrame ("Ejemplo␣1");
24         JButton boton = new JButton ("Haz␣click");
25         final JLabel etiqueta = new JLabel ("Esto␣es␣una␣etiqueta");
26
27         boton.addMouseListener (new MouseAdapter () {
28                 public void mouseClicked (MouseEvent e) {
29                     contador++;
30                     etiqueta.setText ("Clicks:␣"+contador+".");
31                 }
32             });
33
34         JPanel panel = new JPanel ();
35         panel.setBorder (BorderFactory.createEmptyBorder (5,5,5,5));
36         panel.setLayout (new GridLayout (0, 1));
37
38         panel.add (boton);
39         panel.add (etiqueta);
40
41         Container c = frame.getContentPane ();
42         c.add (panel, BorderLayout.CENTER);
43
44         return frame;
45     }
46
47     public static void main (String[] args) {
48
49         Ejemplo1 e = new Ejemplo1 ();
50         JFrame frame = e.dameVentana ();
51
52         frame.pack ();
53         frame.setVisible (true);
54     }
55 }
```

Las clases internas y anónimas fueron pensadas mayormente para interfaces gráficas, pero podemos usarlas cuando y donde queramos. Sin embargo nosotros sólo las utilizaremos para interfaces gráficas (y de hecho es lo recomendable).

En los ejercicios de la práctica podrás utilizar clases normales, internas o anónimas, dependiendo de cómo lo prefieras.

10.3.4. Los componentes de Swing

Swing ofrece varios componentes (herederos casi todos de la clase JComponent) para crear prácticamente cualquier tipo de interfaces gráficas.

Por motivos de espacio veremos sólo una breve descripción de cada uno de los componentes que Swing ofrece, pero descripciones detalladas se darán en el ejemplo completo que se verá más adelante, y sólo para los componentes que se utilicen o que se relacionen con lo que se utilice.

Actividad 10.6 Con el apoyo de tu ayudante, consulta la documentación de la clase JComponent.

Contenedores de primer nivel

Los contenedores de primer nivel son ventanas (excepto Applet), la raíz en nuestra jerarquía de componentes. Todos tienen un contenedor (heredero de la clase Container), llamada *ventana contenedora* o *content pane*. Además de la ventana contenedora, todos los componentes de primer nivel tienen la posibilidad de que se les agregue una barra de menú.

Los contenedores de primer nivel son:

- JFrame. (Marco) Son ventanas autónomas y que no dependen de ninguna otra ventana. Generalmente serán la ventana principal de nuestras aplicaciones.
- JDialog. (Diálogo) Son ventanas más limitadas que los objetos de la clase JFrame. Dependen de otra ventana, que es la *ventana padre* (*parent window*) del diálogo. Si la ventana padre es minimizada, sus diálogos también lo hacen. Si la ventana padre es cerrada, se cierran también los diálogos.
- Applet. (No tiene sentido en español) Emulan a una ventana para correr aplicaciones de Java dentro de un navegador de la WWW. No los veremos.

Para manejar los eventos de un contenedor de primer nivel, se utiliza un *escucha de ventana* o *window listener*.

Actividad 10.7 Con el auxilio de tu ayudante, consulta la documentación de las clase JFrame, JDialog, WindowListener, y WindowAdapter. También consulta la documentación de las clases JOptionPane, y JFileChooser. La clase Applet no se utilizará en estas prácticas.

Contenedores intermedios

Los contenedores intermedios nos sirven para agrupar y presentar de cierta manera distintos componentes. Mientras que se pueden realizar varias cosas con ellos, aquí sólo daremos una breve descripción de cada uno de ellos, y nos concentraremos en los objetos de la clase JPanel en el ejemplo más abajo.

- **Panel.** El contenedor intermedio más flexible y frecuentemente usado. Implementado en la clase JPanel, los páneles no tienen en sí mismos casi ninguna funcionalidad nueva a la que heredan de JComponent. Son usados frecuentemente para agrupar componentes. Un panel puede usar cualquier administrador de trazado, y se le puede definir fácilmente un borde.

- **Ventana corrediza.** Provee barras de desplazamiento alrededor de un componente grande o que puede aumentar mucho de tamaño. Se implementa en la clase JScrollPane.

- **Ventana dividida.** Muestra dos componentes en un espacio fijo determinado, dejando al usuario el ajustar la cantidad de espacio dedicada a cada uno de los componentes. Se implementa en la clase JSplitPane.

- **Ventana de carpeta.** Contiene múltiples componentes, pero sólo muestra uno a la vez. El usuario puede cambiar fácilmente entre componentes. Se implementa en la clase JTabbedPane.

- **Barra de herramientas.** Contiene un grupo de componentes (usualmente botones) en un renglón o en una columna, permitiéndole al usuario el arrastrar la barra de herramientas en diferentes posiciones. Se implementa en la clase JToolBar.

Hay otros tres contenedores intermedios, más especializados:

- **Marco interno.** Es como un marco, y tiene casi los mismos métodos, pero debe aparecer dentro de otra ventana. Se implementa en la clase JInternalFrame.

- **Ventana en capas.** Provee una tercera dimensión, profundidad, para acomodar componentes. Se debe especificar la posición y tamaño para cada componente. Se implementa en la clase LayeredPane.

- **Ventana raíz.** Provee soporte detrás de los telones para los contenedores de primer nivel. Se implementa en la clase JRootPane.

Componentes atómicos

Los componentes atómicos sirven para presentar y/o recibir información del usuario. Son los eventos generados por estos componentes los que más nos interesarán al crear nuestras interfaces gráficas.

Los siguientes son componentes que sirven para recibir información del usuario:

- **Botón, Botón de dos estados, Radio botón.** Proveen botones de uso simple.

  Los botones normales están implementados en la clase JButton.

  Los botones de dos estados son botones que cambian de estado cuando se les hace click. Como su nombre lo indica, tienen dos estados: activado y desactivado. Están implementados en la clase JCheckBox.

  Los radio botones son un grupo de botones de dos estados en los cuales sólo uno de ellos puede estar activado a la vez. Están implementados en la clase JRadioButton.

- **Caja de combinaciones.** Son botones que al hacer click en ellos ofrecen un menú de opciones. Están implementados en la clase JComboBox.

- **Listas.** Muestran un grupo de elementos que el usuario puede escoger. Están implementadas en la clase JList.

- **Menús.** Permiten hacer menús. Están implementados en las clase JMenuBar, JMenu y JMenuItem.

- **Rangos.** Permiten escoger un valor numérico que esté en cierto rango. Están implementados en la clase JSlider.

- **Campo de texto.** Permiten al usuario escribir una sola línea de texto. Están implementados en la clase JTextField.

Los siguientes son componentes que sólo muestran información al usuario, sin recibir ninguna entrada de él:

- **Etiquetas.** Muestran texto, un icono o ambos. Implementadas en la clase JLabel.

- **Barras de progreso.** Muestran el progreso de una tarea. Implementadas en la clase JProgressBar.

- **Pistas.** Muestran una pequeña ventana con información de algún otro componente. Implementadas en la clase JToolTip.

Los siguientes son componentes que presentan información con formato, y una manera de editar esa información:

- **Selector de color.** Una interfaz para seleccionar colores. Implementada en la clase JColorChooser.

- **Selector de archivos.** Una interfaz para seleccionar archivos. Implementada en la clase JFileChooser.

- **Tabla.** Un componente flexible que muestra información en un formato de cuadrícula. Implementada en la clase JTable.

- **Soporte para texto.** Una serie de componentes para manejo de texto, a distintos niveles y con distintos grados de complejidad. Está en las clases JTextArea, JTextComponent, JTextField, y JTextPane.

- **Árbol.** Un componente que muestra datos de manera jerárquica. Implementado en la clase JTree.

10.3.5. Un ejemplo paso a paso

Como se dijo al inicio de la práctica, lo más difícil de las interfaces gráficas es el manejo de eventos. Lo demás es aprender a utilizar los distintos componentes de Swing.

Para que tengan una referencia concreta, veremos un ejemplo completo donde haremos un editor de texto sencillo.

Diseño de un editor de texto sencillo

Queremos hacer un editor de texto sencillo en una clase llamada Editor. Podrá cargar archivos de texto del disco duro, editarlos y salvarlos de nuevo.

La mayor parte de la interfaz gráfica será un componente que nos permita mostrar y editar texto. Además, nuestra interfaz deberá tener un menú de archivo con opciones para abrir, guardar, salir y crear un nuevo archivo, y otro menú de ayuda con información del programa. También deberemos tener una etiqueta que le diga al usuario si el archivo actual ha sido o no modificado.

El componente que nos permite mostrar y editar texto es un objeto de la clase JTextArea, y nuestra ventana principal es un objeto de la clase JFrame. Para acomodar los componentes usaremos un objeto de la clase JPanel.

Ya con esto, nuestra jerarquía de componentes sería como se ve en la figura 10.3.

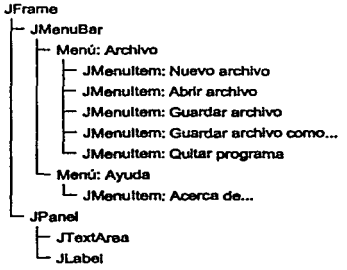


Figura 10.3: Jerarquía de componentes

Variables de clase y clases importadas

Hasta ahora en nuestras clases hemos importado una por una las clases que necesitamos. Java ofrece una manera más cómoda de importar clases. Si hacemos

```
import javax.swing.*;
```

importaremos *todas* las clases del paquete javax.swing. Así que las importaciones en nuestra clase serían

```
3  import javax.swing.*;
4  import javax.swing.event.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.io.*;
8  import java.util.Locale;
```

Usaremos componentes y eventos de Swing, componentes y eventos de AWT, y algunas clases del paquete java.io, para leer y guardar nuestros archivos en disco duro. También importamos la clase Locale, para que el programa hable bien español. Veremos más de esto adelante.

¿Qué variables de clase vamos a necesitar? En primer lugar, nuestro componente de texto deberá ser una variable de clase para que podamos modificarlo de manera sencilla en todos nuestros métodos. La etiqueta para decirle al usuario si se ha modificado el texto también nos conviene hacerla una variable de clase. Y siempre es conveniente hacer nuestra ventana principal una variable de clase.

Usaremos una cadena para saber el nombre del archivo sobre el que estamos trabajando, y un booleano para saber si el texto ha sido modificado. Vamos además a añadir otra cadena para guardar el nombre del archivo (vamos a necesitarlo después), y definamos constantes para el ancho y alto de nuestra ventana:

```
12  public class Editor {
13
14      private JFrame     marco;  // Ventana principal.
15      private JTextArea  texto;  // El texto del archivo.
16      private JLabel     estado; // El estado del archivo.
17
18      private String     archivoOriginal; // El nombre del archivo original.
19      private String     archivo;         // El nombre del archivo actual.
20      private boolean    modificado;
21
22      public static final int BLOQUE = 512;  // Tamaño de bloque de bytes.
23      public static final int ANCHO  = 640;  // Anchura de la ventana.
24      public static final int ALTO   = 480;  // Altura de la ventana.
```

La variable estática BLOQUE [22], la vamos a usar para leer y escribir del disco duro.

Constructores

Tendremos dos constructores; uno sin parámetros, y otro con una cadena como parámetro, que será el nombre de un archivo a abrir. Así podremos iniciar un editor vacío, sin ningún texto, y otro que abra un archivo y muestre su contenido.

```
27  public Editor () {
28      this (null);
29  }
```

```
34  public Editor (String archivo) {
35      modificado = false;
36      archivoOriginal = this.archivo = archivo;
37      creaVentanaPrincipal ();
38      if (archivo != null) {
39          abrirArchivoDeDisco ();
40      }
41      marco.setVisible (true);
42  }
```

Los dos constructores funcionan de manera idéntica; sólo que uno además de inicializar todo, manda abrir un archivo de texto. Para evitar código duplicado, hacemos que el constructor que no recibe parámetros mande llamar al constructor que recibe una cadena [28], y le pase **null** para distinguir cuándo abrimos el archivo.

El constructor inicializa en **false** la variable que nos dice si el archivo está modificado [35], inicializa las variables con el nombre del archivo [36] (que puede ser **null**), y manda crear la ventana principal [37]. Para esto asumimos que tendremos una función abrirArchivoDeDisco que abrirá el archivo especificado por nuestra variable de clase archivo y mostrará su contenido en nuestro componente de texto. Siempre es bueno imaginar que tenemos funciones que hacen ciertas cosas, porque así vamos dividiendo el trabajo en tareas cada vez más pequeñas. Si queremos probar que el programa compila, sólo hay que implementar las funciones vacías.

Si el nombre de archivo que nos pasaron no es **null** [38], mandamos abrir el archivo en disco [39]. Asumimos otro método llamado abrirArchivoDeDisco que (contrario a lo que pudiera pensarse) abrirá el archivo del disco.

Por último, hacemos visible la ventana principal de nuestro programa [41].

Creación de la ventana principal

Veamos ahora cómo crearemos la ventana principal con el método creaVentanaPrincipal:

```
47  public void creaVentanaPrincipal () {
48      marco = new JFrame ("Editor - <ArchivoNuevo>");
49
50      texto = creaAreaDeTexto ();
51      JMenuBar barra = creaBarraDeMenu ();
52      estado = new JLabel ("Hola mundo");
53
54      JPanel panel = new JPanel ();
55      JScrollPane scrollPane = new JScrollPane (texto);
56      scrollPane.setPreferredSize (new Dimension (ANCHO, ALTO));
57
58      panel.setLayout (new BoxLayout (panel, BoxLayout.Y_AXIS));
59      panel.add (scrollPane);
60      panel.add (estado);
61
62      Container c = marco.getContentPane ();
63      c.add (panel);
64
65      marco.setJMenuBar (barra);
66      marco.pack ();
67
68      texto.setRequestFocusEnabled (true);
69      texto.requestFocus ();
70
71      marco.addWindowListener (new WindowAdapter () {
72              public void windowClosing (WindowEvent e) {
73                  menuQuitarPrograma ();
74              }
75          });
76
77      Rectangle medidas = medidasPantalla ();
78      marco.setLocation ((medidas.width-ANCHO)/2, (medidas.height-ALTO)/2);
79  }
```

Primero, instanciamos nuestra ventana principal y le ponemos un título apropiado [48]. Después creamos el área de texto [50], la barra del menú [51], y la etiqueta con el estado del archivo [52]. Aquí volvemos a asumir dos métodos creaAreaDeTexto y creaBarraDeMenu.

Creamos un panel [54] y una ventana corrediza [55]. A ésta le pasamos nuestra área de texto para que esté dentro de ella, y luego le definimos el tamaño con nuestras variables de clase estáticas [56]. Después le ponemos un administrador de trazado de caja [58], el cual sólo acomoda uno tras otro a los componentes, ya sea vertical u horizontalmente. En particular, al nuestro le decimos que los acomode verticalmente (por eso el constructor recibe un BoxLayout.Y_AXIS). Añadimos nuestra ventana corrediza y nuestra etiqueta en el panel [59,60].

Sacamos la ventana contenedora de nuestra ventana principal [62] y le metemos el panel [63]. También le ponemos la barra de menú [65] y empacamos a la ventana [66]. Con esto se ajusta el tamaño de los componentes.

Queremos que cuando el programa empiece el usuario pueda escribir inmediatamente, así que le pasamos el foco a nuestro componente de texto [68,69]. Por último le añadimos un escucha a la ventana para que cuando el evento de cerrar ocurra (widowClosing), entonces el programa termine [71-75]. Asumimos una nueva función: menuQuitarPrograma. El prefijo menu es porque querremos que el evento de cerrar la ventana funcione igual que cuando seleccionemos la opción de quitar el programa del menú; para ambas usaremos este método.

Las dos últimas líneas del método [77,78] obtienen las dimensiones de la pantalla donde esté corriendo el programa, para poder colocar la ventana de forma centrada. Esto es un adorno y es absolutamente prescindible, pero es el tipo de adornos que hacen agradables a los programas. Ahí nos creamos otro método: obtenMedidasPantalla. El método es el que sigue:

```
83  public Rectangle obtenMedidasPantalla () {
84      Rectangle virtualBounds = new Rectangle ();
85      GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment ();
86      GraphicsDevice[] gs = ge.getScreenDevices ();
87      for (int j = 0; j < gs.length; j++) {
88          GraphicsDevice gd = gs[j];
89          GraphicsConfiguration[] gc = gd.getConfigurations ();
90          for (int i = 0; i < gc.length; i++) {
91              virtualBounds = virtualBounds.union(gc[i].getBounds ());
92          }
93      }
94      return virtualBounds;
95  }
```

Está copiado tal cual de la clase GraphicsConfiguration, pero ahí va una explicación rápida: se crea un rectángulo [84], obtenemos el ambiente gráfico [85], del cual sacamos los dispositivos gráficos disponibles [86], a cada uno de los cuales les sacamos la configuración [89], y de todas las configuraciones hacemos una unión [91], para al final regresar el rectángulo [94]. Y así obtenemos las medidas de la pantalla.

Creación del área de texto

Crear el área de texto es relativamente sencillo:

```
100 public JTextArea creaAreaDeTexto () {
101     JTextArea texto = new JTextArea ();
102     texto.setEditable (true);
103     texto.setMargin (new Insets (5, 5, 5, 5));
104     texto.setFont (new Font ("Monospaced", Font.PLAIN, 14));
105     texto.getDocument().addDocumentListener (new DocumentListener () {
106             public void changedUpdate(DocumentEvent e) {
107                 modificado = true;
108                 estado.setText ("Modificado");
109             }
110             public void insertUpdate(DocumentEvent e) {
111                 modificado = true;
```

```
112                 estado.setText ("Modificado");
113             }
114             public void removeUpdate(DocumentEvent e) {
115                 modificado = true;
116                 estado.setText ("Modificado");
117             }
118         });
119     return texto;
120 }
```

Creamos el componente de texto [101], lo hacemos editable [102], le ponemos margen de 5 pixeles alrededor [103], le ponemos una fuente monoespaciada [104] (eso significa que todos los caracteres tienen el mismo ancho), y le ponemos un escucha para que en cada uno de sus eventos, se ponga el estado del editor como "modificado" [105-118] (cambiamos la variable modificado a true, y ponemos la cadena "Modificado" en la etiqueta estado). Esto tiene sentido, ya que los tres eventos implican modificar el texto de alguna manera. Por último regresamos el componente [119].

Creación de la barra de menú

Crear la barra de menú tampoco es difícil:

```
124 public JMenuBar creaBarraDeMenu () {
125     JMenuBar barra = new JMenuBar();
126     JMenu menu;
127     JMenuItem el;
128
129     menu = new JMenu ("Archivo");
130     menu.setMnemonic (KeyEvent.VK_A);
131     barra.add (menu);
132
133     el = new JMenuItem ("Nuevo archivo");
134     el.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK_N,
135                                                ActionEvent.CTRL_MASK));
136     el.addActionListener (new ActionListener () {
137             public void actionPerformed (ActionEvent e) {
138                 menuNuevoArchivo ();
139             }
140         });
141     menu.add (el);
142     el = new JMenuItem ("Abrir archivo");
143     el.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK_O,
144                                                ActionEvent.CTRL_MASK));
145     el.addActionListener (new ActionListener () {
146             public void actionPerformed (ActionEvent e) {
147                 menuAbrirArchivo ();
148             }
149         });
150     menu.add (el);
151     el = new JMenuItem ("Salvar archivo");
152     el.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK_S,
153                                                ActionEvent.CTRL_MASK));
154     el.addActionListener (new ActionListener () {
155             public void actionPerformed (ActionEvent e) {
156                 menuSalvarArchivo ();
157             }
158         });
159     menu.add (el);
160     el = new JMenuItem ("Salvar archivo como...");
161     el.addActionListener (new ActionListener () {
162             public void actionPerformed (ActionEvent e) {
163                 menuSalvarArchivoComo ();
164             }
165         });
```

```
166         menu.add (el);
167         menu.addSeparator();
168         el = new JMenuItem ("Quitar programa");
169         el.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK_Q,
170                                                    ActionEvent.CTRL_MASK));
171         el.addActionListener (new ActionListener () {
172                 public void actionPerformed (ActionEvent e) {
173                     menuQuitarPrograma ();
174                 }
175             });
176         menu.add (el);
177
178         menu = new JMenu ("Ayuda");
179         menu.setMnemonic (KeyEvent.VK_Y);
180         barra.add (menu);
181
182         el = new JMenuItem ("Acerca de...");
183         el.addActionListener (new ActionListener () {
184                 public void actionPerformed (ActionEvent e) {
185                     menuAcercaDe ();
186                 }
187             });
188          menu.add (el);
189
190         return barra;
191     }
```

El método es muy largo; pero hace casi lo mismo seis veces. Primero creamos la barra de menú [125], y después declaramos un menú [126] y un elemento de menú [127]. Vamos a usar dos menús ("Archivo" y "Ayuda") y seis elementos de menú ("Nuevo archivo", "Abrir archivo", "Guardar archivo", "Guardar archivo como...", "Quitar programa" y "Acerca de..."); así que para no declarar ocho variables, declaramos sólo dos y las usamos varias veces.[1]

Primero creamos el menú "Archivo" [129], y le declaramos una tecla mnemónica [130]. La tecla que elegimos es la tecla `a` (por eso el KeyEvent.VK_A), y significa que cuando hagamos `M-a` (o `Alt-a` en notación no-XEmacs), el menú se activará. Y añadimos el menú a la barra [131].

Después creamos el elemento de menú "Nuevo archivo" [133], y le ponemos un acelerador [134,135]. El acelerador es parecido a la tecla mnemónica; cuando lo presionemos, se activará la opción de menú. El acelerador que escogimos es `C-n` (la tecla es KeyEvent.VK_N y el modificador es ActionEvent.CTRL_MASK, o sea `Control`). De una vez le ponemos un escucha al elemento del menú [136-140]. El escucha sólo manda llamar al método (que asumimos) menuNuevoArchivo. Por último, añadimos el elemento de menú al menú [141].

Las líneas [142-176] son una repetición de esto último; creamos los demás elementos de menú, les ponemos aceleradores (no a todos), y les ponemos escuchas que solamente mandan llamar a algún método, ninguno de los cuales hemos hecho. La única línea diferente es la [167], en la que añadimos un separador al menú.

Y por último, en las líneas [178-188] creamos el menú "Ayuda" de forma casi idéntica al menú "Archivo". Lo último que hace el método es regresar la barra de menú [190].

Los eventos del menú

Los eventos del menú son los más importantes, porque serán los que determinen cómo se comporta el programa. Cuando el usuario selecciona un elemento de menú (MenuItem), o

[1] Usar una misma variable para distintos objetos es considerado una mala práctica de programación; pero no es un pecado capital, y puede ser utilizado en casos semi triviales, como éste.

presiona un acelerador designado a él, se dispara un *evento de acción* (*action event*), que se maneja con un *escucha de acción* o *action listener*. La interfaz ActionListener sólo tiene un método, actionPerformed, así que este escucha no tiene adaptador.

Para dividir el trabajo, supusimos que tendríamos un método para cada una de las opciones del menú. Los métodos son: menuNuevoArchivo, menuAbrirArchivo, menuSalvarArchivo, menuSalvarArchivoComo, menuQuitarPrograma, y menuAcercaDe, y va a ser lo siguiente que veamos.

- menuNuevoArchivo

```
196   public void menuNuevoArchivo () {
197       if (modificado) {
198           confirmarDejarArchivo ();
199       }
200
201       texto.setText ("");
202       modificado = false;
203       estado.setText ("␣");
204       archivoOriginal = archivo = null;
205       marco.setTitle ("Editor␣--␣<ArchivoNuevo>");
206   }
```

Lo primero que hace el método es verificar que el archivo en la ventana no esté modificado [197]. Si está modificado, entonces hay que confirmar si se deja o no el archivo [198]; para eso asumimos que tenemos un método confirmarDejarArchivo.

Después limpiamos el texto del componente de texto [201], y dejamos el estado del editor como no modificado [202,203]. Ponemos el nombre del archivo en **null** [204] (porque es nuevo), y cambiamos el título de la pantalla al nuevo estado.

Noten que la variable archivoOriginal no la tocamos.

- menuAbrirArchivo

```
211   public void menuAbrirArchivo () {
212       if (modificado) {
213           confirmarDejarArchivo ();
214       }
215
216       JFileChooser fc = null;
217
218       try {
219           String dir = System.getProperty ("user.dir");
220           fc = new JFileChooser (dir);
221       } catch (SecurityException se) {
222           fc = new JFileChooser ();
223       }
224
225       fc.setDialogTitle ("Abrir␣archivo");
226       int r = fc.showOpenDialog(marco);
227
228       if (r == JFileChooser.APPROVE_OPTION) {
229           File f = fc.getSelectedFile();
230           if (!f.exists()) {
231               JOptionPane.showMessageDialog (marco,
232                                              "El␣archivo␣\""+f+"\"␣no␣existe.",
233                                              "El␣archivo␣no␣existe",
234                                              JOptionPane.ERROR_MESSAGE);
235               return;
236           }
237
238           archivo = f.toString();
239           abrirArchivoDeDisco ();
```

```
240         }
241     }
```

Igual que menuAbrirArchivo, lo primero que hace el método es comprobar si el archivo ha sido modificado [212], y si es así pide confirmación respecto a si se quiere dejar el archivo [213]. Después declaramos un JFileChooser [216].

Un JFileChooser abre una ventana de diálogo donde podemos seleccionar archivos. En un bloque try [218] tratamos de obtener el directorio de trabajo actual (*current working directory*) [219,220] para crear el JFileChooser. Si no podemos [221], usamos el directorio $HOME del usuario [222] (es lo que hace por default el constructor de JFileChooser). Le ponemos título al JFileChooser [225], y obtenemos la opción que haya presionado el usuario del diálogo [226].

Si el usuario presionó la opción "Aceptar" [228], obtenemos el archivo del diálogo [229], y comprobamos que exista [230]. Si el archivo no existe, mostramos un mensaje diciéndolo [231-234] y salimos de la función [235]. Si no entramos al cuerpo del if entonces el archivo existe, así que obtenemos el nombre [238] y lo abrimos con la función abrirArchivoDeDisco [239], que aún no tenemos.

Si el usuario no presionó "Aceptar", entonces presionó "Cancelar", y ya no hacemos nada.

Actividad 10.8 Consulta la documentación de la clase JOptionPane, y presta atención al método estático showMessageDialog. Haz lo mismo con la clase JFileChooser.

- menuSalvarArchivo

```
247     public void menuSalvarArchivo () {
248         if (!modificado) {
249             return;
250         }
251
252         if (archivo == null) {
253             menuSalvarArchivoComo ();
254         } else {
255             salvarArchivoEnDisco ();
256         }
257     }
```

Con este método ocurre lo contrario a los primeros dos; si el archivo no ha sido modificado [248] entonces se sale del método [249].

Después comprueba si el nombre del archivo es null [252]. Si lo es hay que hacer lo que hace el método menuSalvarArchivoComo, y por lo tanto lo manda llamar [253]. Si no, manda salvar el archivo en disco [255], con el método salvarArchivoEnDisco.

- menuSalvarArchivoComo

```
263     public void menuSalvarArchivoComo () {
264         JFileChooser fc = null;
265
266         try {
```

```
267             String dir = System.getProperty ("user.dir");
268             fc = new JFileChooser (dir);
269         } catch (SecurityException se) {
270             fc = new JFileChooser ();
271         }
272
273         fc.setDialogTitle ("Salvar archivo como...");
274         int r = fc.showSaveDialog(marco);
275         File f = fc.getSelectedFile();
276         if (r == JFileChooser.APPROVE_OPTION) {
277             if (f.exists()) {
278                 int r2;
279                 r2 = JOptionPane.showConfirmDialog (marco,
280                                 "El archivo \""+f+"\" existe.\n"+
281                                 "¿Desea sobreescribirlo?",
282                                 "El archivo ya existe",
283                                 JOptionPane.YES_NO_OPTION);
284                 if (r2 != JOptionPane.YES_OPTION)
285                     return;
286             }
287             archivo = f.toString();
288             salvarArchivoEnDisco ();
289         }
290     }
```

En las líneas [264-271] hacemos lo mismo que en el método menuAbrirArchivo: crear un JFileChooser, si es posible con el directorio de trabajo actual. Le ponemos un título adecuado [273], obtenemos la opción que presionó el usuario en el diálogo [274], y obtenemos el archivo que se seleccionó [275].

Si el usuario eligió la opción de "Aceptar" en el diálogo [276], se comprueba que el archivo no exista [277]. Si existe, le preguntamos al usuario si quiere sobreescribirlo [279-283]. Si la respuesta es distinta de "Sí", entonces salimos del método [284,285]. Si no, obtenemos el nombre del archivo [287], y mandamos salvar el archivo [288].

Si el usuario no eligió la opción de "Aceptar", ya no hacemos nada.

- menuQuitarPrograma

```
295     public void menuQuitarPrograma () {
296         if (modificado) {
297             confirmarDejarArchivo ();
298         }
299         System.exit (0);
300     }
```

Si el archivo está modificado [296], pedimos confirmación para dejarlo [297], y después salimos del programa [299].

- acercaDe

```
304     public void menuAcercaDe () {
305         JOptionPane.showMessageDialog (marco,
306                         "Editor de texto sencillo",
307                         "Acerca de...",
308                         JOptionPane.INFORMATION_MESSAGE);
309     }
```

Éste es el método más idiota de todos los tiempos; sólo muestra un diálogo con información del programa (que por cierto no dice nada).

El resto de los métodos

Ya hemos casi terminado los métodos que habíamos supuesto tener. Sólo nos faltan tres:

- abrirArchivoDeDisco

```
317   public void abrirArchivoDeDisco () {
318       char[] caracteres = new char[BLOQUE];
319       int leidos;
320       StringBuffer sb = new StringBuffer (BLOQUE);
321       try {
322           FileReader fr = new FileReader (archivo);
323           do {
324               leidos = fr.read (caracteres);
325               sb.append (caracteres, 0, leidos);
326           } while (leidos != -1 && leidos == BLOQUE);
327           fr.close ();
328           texto.setText (sb.toString());
329           texto.setCaretPosition (0);
330       } catch (IOException fnfe) {
331           JOptionPane.showMessageDialog (marco,
332                       "El archivo \""+archivo+"\" no se pudo leer.",
333                       "Error al leer",
334                       JOptionPane.ERROR_MESSAGE);
335           archivo = archivoOriginal;
336           return;
337       }
338       archivoOriginal = archivo;
339       modificado = false;
340       estado.setText ("");
341       marco.setTitle ("Editor -- "+archivo);
342   }
```

El método primero crea un arreglo de caracteres [318], del tamaño que definimos en la variable BLOQUES, y declara un entero para saber cuántos caracteres leemos en cada pasada [319].

También declara una cadena variable (StringBuffer) [320]. Las cadenas variables son como las cadenas: con la ventaja de que pueden aumentar y disminuir de tamaño, editarse los caracteres que tienen dentro, etc.

Después, dentro de un bloque try [321] abrimos el archivo de texto para lectura [322], y en un ciclo [323-325] leemos todos los caracteres del archivo y los metemos en la cadena variable. Cerramos el archivo [327], ponemos todos esos caracteres como el texto de nuestro componente de texto [328], y hacemos que lo que se vea del texto sean los primeros caracteres [329].

Si algo sale mal en el try [330], asumimos que no pudimos leer el archivo, se lo informamos al usuario [331-334], regresamos el nombre del archivo al original [335] (si no hacemos esto podemos perderlo), y nos salimos de la función [336].

Si salimos airosos del try, actualizamos la variable archivoOriginal [338], y ponemos el estado del editor en no modificado [339,341], lo que tiene sentido pues acabamos de abrir el archivo.

Actividad 10.9 Consulta la documentación de la clase FileReader, y fíjate en los constructores y en los métodos read y close.

- salvarArchivoEnDisco

```
349   public void salvarArchivoEnDisco () {
350       try {
351           FileWriter fw = new FileWriter (archivo);
352           fw.write (texto.getText());
353           fw.close ();
354       } catch (IOException fnfe) {
355           JOptionPane.showMessageDialog (marco,
356                       "El archivo \""+archivo+"\" no se pudo escribir.",
357                       "Error al escribir",
358                       JOptionPane.ERROR_MESSAGE);
359           archivo = archivoOriginal;
360           return;
361       }
362       archivoOriginal = archivo;
363       modificado = false;
364       estado.setText ("");
365       marco.setTitle ("Editor -- "+archivo);
366   }
```

Dentro de un try [350], abrimos el archivo para escritura [351], le escribimos todo el texto de nuestro componente de texto [352], y cerramos el archivo [353].

Si algo sale mal [354], le decimos al usuario que no pudimos salvar su archivo [355-358], regresamos el nombre del archivo al original [359] y salimos del programa [360].

Si salimos bien del try, actualizamos la variable archivoOriginal [362], y ponemos el estado del editor en no modificado [363,365], lo que tiene sentido pues acabamos de salvar el archivo.

Actividad 10.10 Consulta la documentación de la clase FileWriter, y fíjate en los constructores y en los métodos write y close.

- confirmarDejarArchivo

```
371   public void confirmarDejarArchivo () {
372       int r;
373       r = JOptionPane.showConfirmDialog (marco,
374                       "El archivo no se ha salvado.\n"+
375                       "¿Desea salvarlo?",
376                       "El archivo ha sido modificado",
377                       JOptionPane.YES_NO_OPTION);
378       if (r == JOptionPane.YES_OPTION)
379           menuSalvarArchivo ();
380   }
```

Le avisamos al usuario que el archivo está modificado, y preguntamos si quiere salvarlo [373-377]. Si quiere, llamamos al método menuSalvarArchivo, porque es equivalente.

El método main

Por último, hay que ver el método main:

```
385   public static void main (String[] args) {
386       if (args.length > 1) {
387           System.err.println ("Uso: Editor [archivo]");
388           System.exit (1);
389       }
390
391       Locale.setDefault (new Locale ("es", "MX"));
392
393       Editor ed = null;
394
395       if (args.length == 0)
396           ed = new Editor ();
397       else
398           ed = new Editor (args[0]);
399   }
```

Lo que hace main es comprobar que a lo más se llamó al programa con un parámetro [387-390]. Después, define el local del programa para que hable español de México [391], declara un editor [393], y llama al constructor apropiado [395-398].

Actividad 10.11 Consulta la documentación de la clase Locale.

Actividad 10.12 Compila el archivo `Editor.java`. No necesitas especificarle el classpath al compilador. Prueba el editor; tampoco necesitas pasarle nada a la JVM.

Observaciones finales

Hay que reflexionar un poco acerca del diseño de nuestro editor.

Si se dieron cuenta, dentro de los manejadores de eventos (todos en clases anónimas) se intentó utilizar el menor código posible. Poner mucho código en los manejadores de eventos sólo nos complica la vida y hace el código más feo. En general, se encapsula lo que hay que hacer en un manejador de eventos en un solo método, y sólo se invoca.

El diseño se hizo de forma descendiente (*top-down*); comenzamos haciendo los métodos más generales para después hacer los más particulares. Java se presta mucho para trabajar de esta forma, y en el caso de interfaces gráficas nos facilita la vida ya que es muy sencillo pensar en una ventana principal, y qué va a ocurrir cuando elijamos un menú o presionemos un botón.

Es necesario explicar por qué todo el programa consiste de una clase. En el caso de este problema (hacer un editor de texto), todas las clases necesarias las provee Java: toda la parte de interfaz (Swing), y toda la parte de entrada/salida (las clases del paquete java.io). Por lo tanto, sólo teníamos que hacer la clase de uso, que es nuestro programa.

Una última cosa respecto a la clase Locale. Es importante que se acostumbren a tratar de usar el idioma nativo del usuario para un programa. Todas las cadenas que aparecen en clases

como JFileChooser o JMessageDialog están por omisión en inglés. Java por suerte proporciona la habilidad de cambiar dinámicamente esas cadenas, de acuerdo a la lengua que queremos usar.

Java maneja el concepto de *locales*, que es la manera en que determina cómo presentar cierta información al usuario. Esto no sólo se aplica a en qué idioma imprimirá "Sí" o "Archivo", sino a muchas diferencias que hay entre idiomas, y aun entre idiomas iguales en distintos países. Por ejemplo, en México escribimos 1,000.00 para representar un millar con dos cifras decimales; pero en España escriben 1.000,00 para representar lo mismo. Y en ambos países se habla español (o eso dicen en España). Los locales de Java manejan todo este tipo de cosas.

La línea

```
391    Locale.setDefault (new Locale ("es", "MX"));
```

define como local por omisión al que habla español ("es"), en México ("MX"). Pueden comentar esa línea y volver a compilar y ejecutar el programa para que vean la diferencia. Fíjense en particular cuando usamos las clases JFileChooser y JOptionPane.

Los códigos de lenguaje y país usados para determinar los locales están definidos en el estándar ISO-639 e ISO-3166 respectivamente. El primero lo pueden consultar en

<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>

y el segundo en

<http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html>

10.4. Ejercicios

1. Haz una interfaz gráfica para la Base de Datos. La parte principal de la interfaz muestra un registro vacío. Además, debe tener un menú de archivo con las opciones de hacer una Base de Datos nueva, cargar una Base de Datos, guardar una Base de Datos y salir. También debe tener un menú de Base de Datos para dar de alta registros, para darlos de baja, y para buscarlos por nombre y por teléfono. Las opciones de este último menú deben abrir diálogos para realizar sus tareas.

   Toda la interfaz gráfica puede estar contenida sin ningún problema en la clase de uso que hemos venido usando.

   Puedes bajar el código de los paquetes icc1_1.interfaz y icc1_2.interfaz para que te bases en todo el código que veas. También puedes usar a la clase Editor como punto de partida.

10.5. Preguntas

1. ¿Hay alguna parte del editor que no comprendas?
2. ¿Qué se te antoja hacer con interfaces gráficas?

Paquetes y *jarfiles*

Any program will expand to fill available memory.

– Murphy's Laws of Computer Programming #11

Semana:	Decimoquinta semana.
Tiempo de entrega:	Una semana.
Prerrequisitos:	Paquetes.
Objetivo:	Aprender a manejar paquetes.

Esta práctica se realiza durante la decimoquinta semana del curso. El profesor debe haber visto ya paquetes (lo cual debería llevarle a lo más dos clases, y seguramente podrá hacerlo en una).

Ésta es posiblemente la práctica más sencilla de todo el semestre, tanto conceptualmente como en los ejercicios. Los paquetes son útiles para que distintas bibliotecas puedan coexistir sin ningún problema. Conceptualmente son muy sencillos.

Si se ha llegado a este punto en el curso en el tiempo previsto, se ha cubierto la totalidad de los primeros cursos de programación usuales, e incluso se ha abarcado más. Paquetes es un regalo en ese sentido.

Práctica 11
Paquetes y *jarfiles*

11.1. Meta

Que el alumno aprenda a utilizar paquetes y *jarfiles*.

11.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- entender qué son los paquetes,
- crear su propios paquetes, y
- utilizar *jarfiles*.

11.3. Desarrollo

A través de estas prácticas hemos visto cómo hacer programas de diversas complejidades utilizando Java. A medida que la complejidad ha ido en aumento, también ha crecido el número de clases a utilizar; y como crece el número de clases, crece el número de archivos.

Si añadimos además las excepciones, que también deben estar en sus propios archivos, el número de los mismo puede llegar a ser muy, muy grande.

Esta práctica responde dos preguntas que surgen en cuanto nuestro número de archivos crece demasiado; ¿cómo organizamos nuestros programas en Java?, y ¿cómo distribuimos nuestros programas en Java?

11.3.1. Paquetes

Los paquetes son la respuesta que ofrece Java a la organización de código y además a evitar las colisiones de nombre. Una colisión de nombre ocurre cuando dos clases distintas usan el mismo nombre (¿realmente creen que Lista o Registro son nombres muy originales?)

Los paquetes ofrecen otro nivel de encapsulamiento superior a todos los otros que hemos usado; permiten agrupar conjuntos muy diversos de clases.

Además, Los paquetes también nos garantizan una mejor organización de nuestro código fuente, o sea de nuestros archivos `.java`.

Uso de paquetes

Ya hemos usado paquetes en Java en las prácticas anteriores. Los paquetes nos dan un nuevo encapsulamiento y un nuevo tipo de protección para métodos y variables de clase. Por ejemplo, en las prácticas usamos los paquetes icc1_1.interfaz, icc1_1.util, icc1_1.es, icc1_2.interfaz, icc1_2.util e icc1_2.es.

Todas las clases de la biblioteca estándar de Java están agrupadas en distintos paquetes; generalmente, dentro de cada paquete hay clases que comparten cierto comportamiento o utilidad en común.

Para utilizar una clase de un paquete distinto al de nuestro programa, tenemos que *importarla*. Por ejemplo, para usar la clase Consola, necesitábamos hacer al inicio de nuestra clase

```
import icc1_2.interfaz.Consola;
```

Con esto, importábamos la clase Consola nada más. Después vimos que para extraer *todas* las clases del paquete javax.swing, sólo teníamos que hacer

```
import javax.swing.*;
```

No hay ninguna diferencia en el desempeño de un programa si se importa todo un paquete o sólo una clase. El **import** solamente avisa al compilador que debe tener en cuenta ciertos paquetes. Así que da lo mismo cuántas clases se importen.

El único paquete que nunca es necesario avisar que se va a importar es el paquete java.lang. Ahí está la clase Object, por ejemplo. El compilador de Java importa ese paquete siempre.

Dijimos que hay que importar las clases de paquetes *distintos* al de nuestro programa. ¿Qué paquete hemos estado usando? Ninguno explícitamente, y por eso Java asigna un paquete por omisión, que es el que utilizan todas las clases que no especifican un paquete. Esto es importante: *todas* las clases que no definen explícitamente un paquete, pertenecen a un mismo paquete que se llama el *paquete estándar*.

Los paquetes tienen además subpaquetes. Por ejemplo, el paquete javax.swing tiene varios subpaquetes:

- javax.swing.border
- javax.swing.event
- javax.swing.plaf
- javax.swing.text
- javax.swing.undo
- javax.swing.colorchooser
- javax.swing.filechooser
- javax.swing.table
- javax.swing.tree

(Y por cierto, hacer **import** javax.swing.*.*; *no* funciona para traer a todas las clases del paquete y además todas las clases de sus subpaquetes).

Un subpaquete no tiene nada de particular, sólo es para especificar un subconjunto de clases de un paquete que son lo suficientemente autocontenidas como para estar en un mismo paquete, pero que siguen teniendo muchas similitudes con el paquete principal, o que dependen de él.

Los nombres de los paquetes son arbitrarios, pero por supuesto lo mejor es que tengan cierto sentido. Además, se sigue la convención de que las compañías usen su dominio de Internet invertido como prefijo de sus paquetes. La compañía con dominio <http://www.acme.com.mx>, por ejemplo, utilizaría paquetes cuyos nombres comenzarían con mx.com.acme. Sun Microsystems (la compañía que diseñó Java) por ejemplo usa paquetes que comienzan con com.sun.

Creación de paquetes

Para definir que una clase es de un paquete, sencillamente hay que incluir

```
package mipaquete.misubpaquete;
```

al inicio de una clase; preferentemente antes de importar clases o paquetes enteros (para mayor claridad). Una clase sólo puede pertenecer a un único paquete.

Hasta aquí no hay mayor problema. Los problemas se presentan cuando tratemos de compilar un archivo así.

Los paquetes no sólo nos permiten agrupar nuestras clases de forma abstracta, también son una forma de agrupar nuestras clases *concretamente*, ya que deben estar en directorios que concuerden con el nombre del paquete. Si tenemos una clase que queremos que sea del paquete mipaquete.misubpaquete, entonces el archivo con la clase debe estar en un directorio llamado `misubpaquete` que a su vez esté en otro directorio llamado `mipaquete`, y para compilarla deberemos hacerlo desde el directorio padre de `mipaquete`.

Esto nos genera una jerarquía de directorios,[1] que nos permite tener separadas y ordenadas nuestras clases. Por ejemplo, los paquetes que usamos durante la primera parte del curso utilizaban la jerarquía de archivos directorios que se ve en la figura 11.1.

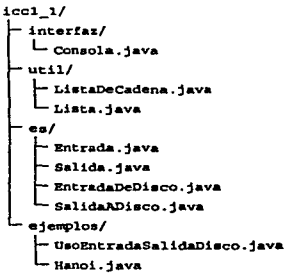


Figura 11.1: Jerarquía de archivos y directorios del paquete icc1_1

(No mostramos la jerarquía de icc1_2 porque con todas las excepciones el número de archivos crece bastante).

Supongamos que tenemos la clase mipaquete.misubpaquete.MiClase. Ya dijimos que para empezar necesita estar en el subdirectorio `misubpaquete` del directorio `mipaquete`. Para compilarla, nos subimos al directorio padre de `mipaquete`, y hacemos

```
# javac mipaquete/misubpaquete/MiClase.java
```

Para ejecutarla, desde ese mismo directorio hacemos

[1] Java tiene *muchas* jerarquías.

```
# java mipaquete.misubpaquete.MiClase
```

Es válido también hacer

```
# java mipaquete/misubpaquete/MiClase
```

pero se recomienda la primera forma, ya que es portátil entre distintas plataformas.

El acceso de paquetes

Debe ser obvio después de todo esto que el acceso de paquete (package) es menos restrictivo que protected pero más restrictivo que public. Cuando una variable de clase o método tiene acceso package, sólo puede ser visto o utilizado por métodos dentro del mismo paquete.

El *classpath*

¿Qué pasa si no queremos compilar o ejecutar precisamente en ese directorio a nuestra clase? Entonces nada más le especificamos con el *classpath* en dónde buscar clases

```
# javac -classpath <directorioPadreDeMiPaquete> mipaquete/misubpaquete/MiClase.java
# java -cp <directorioPadreDeMiPaquete> mipaquete.misubpaquete.MiClase
```

Si tenemos distintos paquetes en distintos directorios, sólo los separamos con dos puntos (:). Si estamos usando el compilador de Sun (el que viene con el JDK), por omisión, no hay necesidad de especificar el *classpath* de los paquetes incluidos con Java; ya sabe dónde están y cómo buscarlos, por lo que el compilador y la JVM se encargan de todo.

Esto no es del todo cierto si usamos algún otro compilador de Java. Hay muchos compiladores de Java por ahí en la red.

JavaDoc y los paquetes

Para generar documentación de JavaDoc para un paquete, por ejemplo del paquete mipaquete.misubpaquete, sólo hay que colocarse en el directorio padre del directorio `mipaquete`, y hacer

```
# javadoc mipaquete.misubpaquete
```

Esto generará la documentación de todas las clases del paquete. Si queremos poner una descripción del paquete en la documentación generada (el *overview*), sólo agregamos un archivo llamado `package.html`, con formato HTML, en el directorio `mipaquete/misubpaquete/`.

11.3.2. *Jarfiles*

Supongamos ahora que ya tenemos un programa enorme con cientos de clases, dividido en varios paquetes, que además utiliza otras varias decenas de clases distribuidas en otras varias decenas de paquetes.

Nuestro programa ya compila y corre y, orgullosos, queremos decirle al mundo de él y presumirlo con pompa y circunstancia. Así que ponemos una página en la WWW y pedimos a la gente que baje y pruebe nuestro programa.

No podemos decirles que bajen todos nuestros archivos `.class`, y que después los acomoden en la jerarquía de directorios necesaria (de hecho sí podríamos decirles eso, otra cosa es que lo hicieran). Tenemos que encontrar la manera de distribuir nuestras clases y paquetes de una manera sencilla y eficiente. Para esto están los *jarfiles*.

Uso y creación de *jarfiles*

Ya hemos usado *jarfiles* también. Durante todo el curso, varias clases (de hecho varios paquetes) se les distribuyeron como *jarfiles*. Un *jarfile* es sólo un archivo con formato de compresión ZIP, que incluye todas las clases de uno o varios paquetes, organizadas en los directorios correspondientes. Además, un *jarfile* puede contener imágenes o sonidos necesarios para que un programa se ejecute como debe ser.

Como el *jarfile* contiene dentro de sí una estructura de directorio, sólo hay que pasárselo al *classpath* del compilador o de la JVM para utilizarlo, como hicimos a lo largo del curso

```
# java -cp paquete.jar mipaquete.paquete.MiClase
```

Para crear un *jarfile*, por ejemplo para el paquete mipaquete.misubpaquete, sólo nos ponemos en el directorio padre de `mipaquete` y hacemos

```
# jar cf mipaquete.jar mipaquete/misubpaquete/*.class
```

El comando `jar` es el utilizado para generar los *jarfiles*.

Actividad 11.1 Consulta la página del manual de `jar` haciendo

```
# man jar
```

En pocas palabras, la `c` es para decirle que cree el *jarfile*, y la `f` es para decirle cómo se llamará el archivo.

Java tiene un paquete especializado para tratar con *jarfiles*. Es el paquete java.jar.

Actividad 11.2 Consulta la documentación de las clases en el paquete java.jar.

11.4. Ejercicios

1. Empaqueta las clases de tu Base de Datos como mejor te parezca. No es necesario que uses subpaquetes. Asegúrate de que tu programa siga compilando y de que puedas generar la documentación de JavaDoc.

Con este ejercicio terminamos con nuestra Base de Datos en el curso. Por supuesto, no es una Base de Datos en todo el sentido del término (seguramente llevarán un curso completo de Bases de Datos más adelante en la carrera), pero varias de las cosas que hemos visto se aplican.

Sólo queremos especificar algo respecto a las Bases de Datos reales que ignoramos a lo largo de todas estas prácticas, porque *sí* es un concepto importante. Durante todo el curso manejamos la idea de que una Base de Datos es una tabla. En la práctica 5 cambiamos la representación interna con una lista, pero conceptualmente podíamos seguirla viendo como una tabla (cada registro como un renglón, cada campo como una columna).

Las Bases de Datos reales son un *conjunto* de tablas, organizadas alrededor de relaciones entre las columnas de las tablas.

11.5. Preguntas

1. ¿Tienen sentido los paquetes? Justifica.

Hilos de Ejecución y Enchufes (Opcional)

The value of a program is proportional to the weight of its output.

– Murphy's Laws of Computer Programming #12

Semana:	Decimosexta semana.
Tiempo de entrega:	El máximo posible.
Prerrequisitos:	Hilos de ejecución.
Objetivo:	Aprender a manejar hilos de ejecución y enchufes.

Esta práctica se realiza durante la decimosexta semana del curso. El profesor debe haber visto ya hilos de ejecución, en lo que podría aprovechar la última semana del curso.

Los hilos de ejecución es un tema que tradicionalmente no se enseña en el primer curso de programación. Por lo mismo esta práctica es opcional, para ganar puntos extras, y sólo aquellos que quieran/puedan hacerla la harán (lamentablemente, es muy probable que sólo aquellos que no la necesiten la hagan).

Los hilos de ejecución no son muy complicados, y Java los maneja de manera bastante sencilla para el poder que proporcionan. Enchufes es entrada y salida a larga distancia; no deberían tener problemas.[2]

Los problemas van a ocurrir cuando junten todo. Pero justamente por eso esta práctica es opcional. Es un reto para aquellos que quieran aceptarlo. El tiempo de entrega es abierto, lo que quiere decir para que tienen hasta que se de por terminado el curso para que entreguen la práctica.

[2]Sí, el término *enchufe* puede no ser del gusto de algunas personas, pero es la traducción correcta al término *socket* en inglés.

Práctica 12
Hilos de Ejecución y Enchufes (Opcional)

12.1. Meta

Que el alumno aprenda a utilizar hilos de ejecución y a programar en red con enchufes.

12.2. Objetivos

Al finalizar esta práctica el alumno será capaz de:

- entender y utilizar hilos de ejecución, y
- entender y utilizar enchufes.

12.3. Desarrollo

A lo largo de las prácticas se ha cubierto el material para Java de un primer curso de programación en Ciencias de Computación.

Queremos sin embargo cubrir dos puntos que, aunque probablemente algo avanzados para un primer curso de programación, creemos es necesario mencionarlos y discutirlos, ya que en la actualidad son conocimientos obligatorios para cualquier computólogo.

El propósito de esta última práctica es mencionar los hilos de ejecución y la programación en red con enchufes.

12.3.1. Hilos de Ejecución

La ejecución de nuestros programas, hasta la práctica 10, era lineal. Se invocaba al método main de nuestra clase de uso, y a partir de ahí se invocaba al método A, B, C, etc., de distintas clases. Dentro de cada uno de esos métodos se podía a la vez llamar a más métodos (y en la práctica 8 vimos lo que ocurría cuando un método se llamaba a sí mismo). Y desde la práctica 5 sabemos que dentro de un método puede haber uno o varios ciclos dando vueltas. Incluso podemos tener ciclos dentro de ciclos.

Pero al fin y al cabo, si nuestras funciones recursivas están bien hechas y nuestros ciclos terminan algún día, el programa continuará su ejecución tranquilamente, *una instrucción a la vez.*

Las aplicaciones modernas dejaron de funcionar así hace ya mucho tiempo. Uno tiene editores como XEmacs donde se puede compilar un programa en un buffer mientras se edita un archivo en otro buffer *al mismo tiempo.* O navegadores como Mozilla que pueden abrir varias páginas de la red al mismo tiempo, y algunas de esas páginas pueden reproducir música o video, y además el navegador puede estar bajando varios archivos a la red, *todo al mismo tiempo.*

Por supuesto, para que todo esto funcione, el sistema operativo debe proveer de la funcionalidad necesaria. Para esto, hace ya casi cuarenta años surgió el concepto de *multiproceso*, que es lo que permite que hagamos varias cosas en la computadora al mismo tiempo.

A pesar de que en la actualidad ya no es tan raro encontrar computadoras personales con dos o incluso cuatro procesadores, la gran mayoría de las computadoras de escritorio siguen contando con un único procesador. Un procesador sólo puede ejecutar una instrucción a la vez. Los procesadores actuales pueden ejecutar varios millones de instrucciones en un segundo, pero una a una. Incluso en las computadoras con múltiples procesadores, cada procesador sólo puede ejecutar una instrucción a la vez.

Los sistemas operativos reparten el procesador entre varios procesos (aunque algunos lo hacen mucho peor que otros). Como los procesadores son muy rápidos (y cada vez lo son más), no se nota que en cada instante dado, el procesador sólo se hace cargo de un proceso.

Gracias a la capacidad multiproceso de los sistemas operativos, los lenguajes de programación implementan un comando para que el proceso de un programa pueda *dividirse* en dos procesos (proceso padre y proceso hijo), cada uno de los cuales puede seguir caminos muy distintos. La instrucción suele llamarse *fork*.

El problema de dividir procesos es que cuando se hace, toda la imagen del proceso padre en memoria se copia para el proceso hijo. Además, son procesos completamente independientes; si el proceso padre abre archivos por ejemplo, no puede compartirlos con el proceso hijo.

Imagínense un programa como Mozilla o el Internet Explorer. Son programas grandes y ocupan mucho espacio en memoria. Cuando el usuario solicita que el navegador comience a bajar un archivo de la red, queremos que pueda seguir navegando mientras el archivo se descarga. Si dividimos el proceso, *toda* la imagen en memoria de Mozilla debe copiarse, cuando lo único que queremos es la parte del programa que se encarga de bajar archivos. Y toda esa memoria tiene que liberarse una vez que el archivo se haya descargado. Además, imaginen que al usuario se le ocurre bajar siete archivos al mismo tiempo (a algún usuario se le va a ocurrir que eso es una buena idea...).

En vista de lo costoso que resulta dividir procesos, surgió el concepto de *procesos ligeros* o *hilos de ejecución* (*threads* en inglés).[1]

Un hilo de ejecución es como un proceso hijo; pero se ejecuta en el contexto de su proceso padre. Pueden compartir variables, intercambiar información, y lo que muchos consideran lo más importante: *no se copia toda la imagen* en memoria del proceso padre.

Aunque es posible que no se hayan dado cuenta, nosotros ya hemos usado varios hilos de ejecución. Al momento de empezar a utilizar eventos, comenzamos a utilizar varios hilos de

[1]Los hilos de ejecución ya existían en Algol extendido, lenguaje de programación que usaban las máquinas Burroughs (hoy Unisys) en 1970. Todos los sistemas Unix, que están basados en el estándar POSIX, utilizan los Posix Threads implementados en el leguaje de programación C, desde hace ya varios años. El concepto no es nuevo, pero comenzó a popularizarse hasta hace relativamente pocos años.

ejecución en un programa.

Cuando tenemos un programa orientado a eventos (como los son la gran mayoría de los programas que utilizan una interfaz gráfica para comunicarse con el usuario), la parte que espera que los eventos ocurran se ejecuta en un hilo de ejecución distinto al del programa principal.

Fíjense en el método main de la clase Editor (o el de su propia interfaz gráfica para su Base de Datos). Después de crear el objeto de la clase Editor con new, el programa principal termina. Ya no se ejecuta ningún método o instrucción; main sencillamente acaba. Ese hilo de ejecución *termina* (porque aunque usemos un único hilo de ejecución en un programa, éste sigue siendo un hilo de ejecución).

La ejecución del programa continúa porque hicimos visible un objeto de la clase JFrame con el método setVisible. En ese momento comienza a ejecutarse otro hilo de ejecución que pueden imaginarse como el siguiente código:

```
Event e;
do {
    e = eventoActual ();
    if (e != null) {
        /* Ejecuta todos los manejadores del evento */
    }
} while (true);
```

Por supuesto, no es así exactamente, pero ésa es la esencia. El hilo de ejecución de los componentes gráficos de Java es un ciclo infinito, que lo único que hace es detectar qué eventos ocurren y ejecutar los manejadores correspondientes. Si lo piensan tiene sentido; en un programa con interfaz gráfica (XEmacs, Mozilla, nuestro editor, etc.), si ustedes no hacen nada el programa tampoco. Se queda esperando hasta que hagan algo para reaccionar de forma correspondiente. Se queda esperando en un ciclo que nunca termina.

Además todos los manejadores de eventos se ejecutan en *otro* hilo de ejecución. Así que cuando trabajamos con interfaces gráficas en Java, siempre se están usando tres hilos de ejecución.

Hilos de Ejecución en Java

Por sorprendente que resulte, en Java los hilos de ejecución son clases.

Actividad 12.1 Consulta la documentación de la clase Thread y de la interfaz Runnable.

Un hilo de ejecución es un objeto de la clase Thread, o de alguna clase que la extienda. Antes de que usáramos hilos de ejecución, si un método A tenía este cuerpo:

```
{
    // Instrucción 1
    B (); // Llamamos al método B.
    // Instrucción 2
}
```

lo que ocurría al entrar al método era:

1. Se ejecutaba la instrucción 1.
2. Se llamaba al método B (ejecutándose todas las instrucciones, ciclos, recursiones, etc. del método).
3. Se ejecutaba la instrucción 2.

Con un hilo de ejecución, en cambio, tenemos esto (si suponemos que tenemos un hilo de ejecución llamado t):

```
{
    // Instrucción 1
    t.start (); // Llamamos al método start del hilo de ejecución.
    // Instrucción 2
}
```

En este caso, el método start se ejecuta *al mismo tiempo* que la instrucción 2. El método start *regresa inmediatamente* después de haber sido llamado, y ahora la ejecución del programa corre en dos hilos de ejecución paralelos.

En los dos hilos de ejecución podemos hacer cosas, y los hilos de ejecución pueden verse y hablarse (pasarse objetos y tipos básicos).

Después de todo lo que se dijo al inicio de la sección, ustedes *saben* que los dos hilos de ejecución no se ejecutan al mismo tiempo exactamente. Pero la JVM, con la ayuda del sistema operativo, se encarga de que parezca que sí se ejecutan al mismo tiempo.

Creación de hilos de ejecución

Para crear un hilo de ejecución, necesitamos extender la clase Thread:

```
public class MiProceso extends Thread {
    ...
```

Dentro de la clase MiProceso lo que hacemos es sobrecargar el método run, que no recibe ningún parámetro y tiene tipo de regreso void:

```
    public void run () {
        // Aquí implementamos nuestro hilo de ejecución.
    }
```

Después, cuando queramos ejecutar el hilo de ejecución, llamamos al método start que se hereda de la clase Thread, y start a su vez ejecuta el método run.

```
public class UsoMiProceso {
    public static void main (String[] args) {
        MiProceso mp = ... // Instanciamos el hilo de ejecución.
        ...                // Hacemos más cosas.
```

```
mp.start ();          // Ejecutamos el hilo de ejecución.
...                   // Hacemos todavía más cosas.
```

No llamamos directamente a run justamente porque start se encarga de hacer lo necesario para que run se ejecute en su propio hilo de ejecución. El método start regresa inmediatamente, y por lo tanto la ejecución del programa ahora sigue dos caminos (dos hilos de ejecución): las expresiones que siguen después de llamar a start, y las expresiones dentro del método run de nuestro objeto de la clase MiProceso. Dentro de cualquiera de los dos hilos de ejecución se pueden hacer llamadas a distintos métodos, recursión, ciclos, etc. Todo esto se vería gráficamente como en la figura 12.1

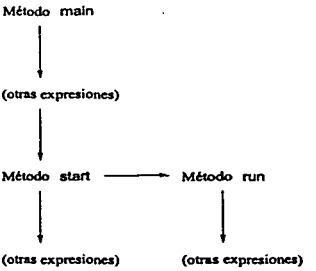


Figura 12.1: Ejecución de un hilo de ejecución

Noten que no es obligatorio que el método start sea llamado desde main; puede ser llamado desde cualquier lugar en el programa (por eso ponemos que hay otras posibles expresiones desde que entramos a main hasta que llamamos a start).

Extender a la clase Thread siempre que queramos un hilo de ejecución independiente puede resultar restrictivo. Tal vez en el diseño de nuestro problema nos encontremos con una clase que está dentro de una jerarquía de herencia específica, y que además tiene que ejecutarse en su propio hilo de ejecución.

Para esto está la interfaz Runnable. Cuando una clase que hagamos implemente la interfaz Runnable, tiene que definir el método run (que es el único método declarado en la interfaz):

```
public class MiProceso extends AlgunaClase implements Runnable {
    public void run () {
        ...
    }
    ...
```

Y para ejecutar nuestro hilo de ejecución, creamos un objeto de la clase Thread *utilizando* el objeto de nuestra clase. Con este nuevo objeto podemos ya llamar al métod start:

```
public class UsoMiProceso {
    public static void main (String[] args) {
        MiProceso mp = ...    // Instanciamos el objeto runnable.
        ...                   // Hacemos más cosas.
        Thread t;             // Declaramos un hilo de ejecución, y
        t = new Thread (mp);  // lo instanciamos con nuestro objeto.
        t.start ();           // Ejecutamos el hilo de ejecución.
        ...                   // Hacemos todavía más cosas.
```

En el ejemplo pusimos que la clase MiProceso extiende a la clase AlgunaClase. Esto es porque sólo hay que implementar la interfaz Runnable si nuestra clase tiene que heredar a alguna otra que no sea Thread. Si nuestra clase no necesita extender a ninguna otra, lo correcto es que extienda a Thread.

Vida y muerte de los hilos de ejecución

Hay que tener bien claro cómo empiezan, cómo terminan, y cómo se ejecutan los hilos de ejecución. Un hilo de ejecución sólo comienza cuando, con el método start, se llama al método run. Cuando creamos un nuevo hilo de ejecución con new, el hilo de ejecución se considera *vacío*: la JVM no ha hecho todavía nada con él para que se pueda ejecutar en paralelo. Si llamamos a cualquier método del hilo de ejecución *antes* de que llamemos al método start, la JVM lanzará la excepción IllegalThreadStateException.

Una vez que llamamos al método start, la JVM hace todo lo que se necesita para que el hilo de ejecución se ejecute. Esto implica comunicarse con el sistema operativo, asignar memoria, etc., y entonces manda llamar al método run del hilo de ejecución.

El hilo de ejecución termina cuando sale del método run. Dentro de este método puede hacer recursiones, mandar llamar otros métodos, hacer ciclos, y todo lo que se hace en un método normal; pero tarde o temprano (si el método está bien hecho) termina, y entonces el hilo de ejecución termina también, con lo que se liberan todos los recursos de la computadora que pudiera haber utilizado. Por ello generalmente tendremos un ciclo en el método run (o llamaremos a un método del hilo de ejecución que tenga un ciclo), que continuará hasta que algo ocurra o deje de ocurrir.

Mientras el hilo de ejecución esté corriendo, podemos hacer varias cosas con él. La más importante de ellas es sincronizarlo con otros hilos de ejecución, pero veremos esto un poco más adelante. Otra cosa que podemos hacer es ponerlo a dormir durante un determinado número de milisegundos:

```
try {
    Thread.sleep (1000);
} catch (InterruptedException e){
    // No podemos dormir ahora, así que seguimos.
}
```

El método estático sleep de la clase Thread tratará de poner a dormir el hilo de ejecución donde se manda llamar el método (por eso es estático). El método espera de la clase GraficosDeReloj de las primeras dos prácticas es lo que hacía.

Puede ocurrir que la JVM decida que el hilo de ejecución no puede dormirse en ese momento, por lo que el método lanza la excepción InterruptedException cuando eso ocurre. Realmente no hay mucho que hacer en esos casos; sencillamente el hilo de ejecución continuará su ejecución sin dormirse.

Otra cosa que podemos hacer es preguntar si un hilo de ejecución está vivo o no:

```
if (mp.isAlive ()) {
    // El hilo de ejecución está vivo.
}
```

Este método se invoca desde *afuera* del hilo de ejecución que nos interesa, no adentro. Si un hilo de ejecución está vivo quiere decir que ya fue llamado su método run, y que éste no ha terminado.

Además de esto, podemos cambiar la *prioridad* de un hilo de ejecución. La prioridad de un hilo de ejecución es la importancia que le da la JVM para ejecutarlo. Como discutimos arriba, una computadora generalmente tiene un procesador, y en algunos casos dos o cuatro. La JVM (junto con el sistema operativo) tiene que repartir el procesador entre todos los hilos de ejecución que existan. En principio, trata de ser lo más justa que puede, asignándole a cada hilo de ejecución relativamente la misma cantidad de tiempo en el procesador.

Podemos cambiar eso con el método setPriority. Sin embargo, no es un método confiable en el sentido de que no va a funcionar igual entre arquitecturas distintas, o entre sistemas operativos diferentes. Pueden realizar experimentos con el método, pero realmente se recomienda no basar el funcionamiento de un programa en las prioridades de sus hilos de ejecución.

Sincronización de hilos de ejecución

En los ejemplos vistos hasta ahora, hemos visto los hilos de ejecución como tareas totalmente independientes ejecutándose al mismo tiempo. La mayor parte de las veces, sin embargo, querremos que los hilos de ejecución compartan información o recursos.

Esto trae problemas, por supuesto. Los hilos de ejecución deben ponerse de acuerdo en cómo van a tener acceso a la información o recursos para que no se estorben mutuamente.

Hay un ejemplo bastante viejo, que hace mención a cinco filósofos sentados alrededor de una mesa circular. Enfrente de cada filósofo hay un tazón de arroz, y a la derecha de cada filósofo hay un palillo.

Para comer un bocado de arroz, cada filósofo necesita el palillo que le corresponde, y además el palillo a su izquierda (que le toca a otro filósofo). Si todos los filósofos agarran su palillo al mismo tiempo, ninguno va a poder comer. Es necesario que se pongan de acuerdo de alguna manera para que algunos puedan agarrar dos palillos mientras otros esperan, y después ir cambiando turnos.

El ejemplo es de sistemas operativos, ya que sirve para ejemplificar de manera muy sencilla lo que pasa cuando varios procesos quieren usar el procesador (o la memoria, o el disco duro). El caso en que el sistema se traba porque todos los procesos quieren usar el mismo recurso al mismo tiempo (como cuando los filósofos se quedaban cada uno con nada más un palillo) se le llama *abrazo mortal* o *deadlock*.

Para evitar los abrazos mortales, lo primero que debemos hacer es evitar que dos hilos de ejecución distintos traten de llamar al mismo método al mismo tiempo. Para esto hacemos que

el método esté *sincronizado*, lo que se logra agregando la palabra clave **synchronized** en la definición del método:

```
public synchronized void metodo () {
    ...
```

Esto se hace en los métodos de los objetos que vayan a ser compartidos por los hilos de ejecución, no en los métodos de los hilos de ejecución mismos (se puede hacer también, pero realmente no tiene mucho sentido). Casi todas las clases de Java tienen sincronizados sus métodos.

Además de sincronizar métodos completos, podemos sincronizar un bloque respecto a una o varias variables (si asumimos que queremos sincronizar respecto al objeto obj):

```
synchronized (obj) {
    obj.metodo ();
    obj.defineVariable (5);
}
```

Cuando el hilo de ejecución llegue a esa parte del código, ningún otro hilo de ejecución podrá hacer uso del objeto obj.

La segunda cosa que debemos hacer para evitar abrazos mortales es hacer que un hilo de ejecución espere hasta que le avisen que ya puede tener acceso a algún recurso u objeto. Por ejemplo, en nuestro ejemplo de los filósofos podemos hacer que los filósofos 2 y 4 esperen a que los filósofos 1 y 3 hayan usado los palillos para que ellos los usen, y que el filósofo 5 espere a los filósofos 2 y 4, y por último que los filósofos 1 y 3 esperen a que el 5 acabe para continuar.[2]

Para que un hilo de ejecución espere hasta que le avisen que ya puede seguir su ejecución, está el método wait:

```
try {
    // espearamos a que nos avisen que podemos continuar.
    wait();
} catch (InterruptedException e) {
}
```

El método wait está definido en la clase Object, así que todos los objetos de Java pueden llamarlo. El método lanza la excepción InterruptedException si por algún motivo no puede esperar. Además, en la clase Object están definidas otras dos versiones del método wait, que sirven para que el hilo de ejecución sólo espere un determinado número de milisegundos (o nanosegundos, si el sistema operativo puede manejarlos).

El método notifyAll (también definido en la clase Object) hace que todos los hilos de ejecución que estén detenidos por haber llamado al método wait continúen su ejecución.

Con estos dos métodos podemos hacer que los hilos de ejecución se pongan de acuerdo en cuándo deben tener acceso a algún recurso u objeto, para que no traten de hacerlo todos al mismo tiempo.

[2]Ésta es una solución que garantiza que todos los filósofos comen, pero es ineficiente porque mientras el filósofo 5 come podría hacerlo también algún otro filósofo. Hay muchas soluciones distintas para el problema de los filósofos y los palillos.

Grupos de hilos de ejecución

Nada más para no dejar de mencionarlo, cada hilo de ejecución en Java pertenece a un grupo de hilos de ejecución. Un hilo de ejecución que no especifique a qué grupo pertenece se asigna al grupo de hilos de ejecución por omisión.

A un grupo de hilos de ejecución se les puede tratar como conjunto, lo que permite que los pongamos a dormir, o a esperar a todos al mismo tiempo. Todo lo demás que tenga que ver con grupos de hilos de ejecución queda fuera del alcance de esta práctica.

Diseño con hilos de ejecución

Hay que entender que los hilos de ejecución son clases de uso de alguna manera. Son tareas que nuestro programa debe realizar al mismo tiempo que realiza la tarea "principal" de nuestro programa (entre comillas porque todos los hilos de ejecución, por lo menos por omisión, tienen la misma prioridad unos sobre otros). Las clases que extendamos de Thread o en que implementemos Runnable no se mapearán con algún elemento concreto de nuestro problema: se van a mapear con una *tarea* específica de nuestro programa, que no puede realizarse antes o después de la tarea principal, sino que tiene que realizarse al mismo tiempo.

Piensen que hacemos un programa que representa un restaurante. Nuestras clases obvias serían las que representaran al cocinero, a los meseros, al capitán y a los comensales. Pero además tenemos que hacer que el cocinero haga los platillos, que los meseros solicitan órdenes, que el capitán reciba más comensales, y que los clientes coman, todo al mismo tiempo. Para esto hacemos que cada una de esas tareas sea un hilo de ejecución.

Hay que tener en cuenta los abrazos mortales cuando se diseña un programa con varios hilos de ejecución. Los objetos que vayan a ser compartidos por varios hilos de ejecución es recomendable que sus métodos sean sincronizados, y debe hacerse un análisis profundo del problema para ver si es necesario utilizar los métodos wait y notifyAll. Habrá ocasiones en que podrá evitarse.

Hemos mencionado varias veces objetos compartidos por los hilos de ejecución, y que éstos pueden hablarse y verse. El método run no recibe parámetros, así que ¿cómo le pasamos información a los hilos de ejecución?

Por suerte, los hilos de ejecución son clases, y además clases que nosotros mismos definimos. Por lo tanto podemos simplemente usar el constructor de nuestra clase que extienda a Thread para pasarle objetos, o conjuntos de objetos (usando listas o arreglos).

12.3.2. Programación en red

Desde hace varios años, las aplicaciones que trabajan sobre la red han aumentado en número e importancia. Hoy en día casi cualquier profesionista necesita de un navegador y de un cliente de correo electrónico para trabajar.

La programación en red, también llamada de diseño cliente/servidor, es un tema en sí mismo amplio y fuera del alcance de estas prácticas. Sin embargo, se les dará una pequeña muestra de cómo funciona para que vean los fundamentos básicos de este tipo de programas.

Para crear programas que funcionen sobre la red usando Java hay varios métodos, de los cuales los más conocidos son:

- *Enchufes*, que es la forma tradicional de escribir programas de diseño cliente/servidor, y que tienen su propia biblioteca en Java. Los *enchufes* (*sockets* en inglés) envían y reciben bytes, o sea que funcionan a bajo nivel.

- *RMI*, o *Remote Method Invocation*, (*Invocación Remota de Métodos*). Es la versión en Java del *RPC* del lenguaje de programación C (*Remote Procedure Call* o *Llamado Remoto de Procedimientos*). Consiste en transmitir objetos completos a través de la red de una máquina a otra, donde pueden ejecutar sus métodos.

- Servlets y JSP. JSP significa *Java Servlet Page*, y funciona muy similarmente a PHP o las páginas ASP de Microsoft (PHP significa *PHP: Hypertext Preprocessor* mientras que ASP se entiende por *ActiveX Server Page*). Los servlets son programas de Java normales que se ejecutan en una máquina que funciona como servidor de WWW, y que generan páginas dinámicamente. Los JSP, así como PHP y las ASP funcionan similarmente; pero en lugar de ser programas normales, es código incrustado dentro de una página HTML. Todos estos métodos son utilizados para crear programas cuyos clientes necesitan un navegador para ejecutarse, como Mozilla o el Internet Explorer.

- CORBA. CORBA es un sistema de ejecución de programas distribuido, que excede por mucho en complejidad, poder y teoría a cualquiera de los anteriores. La meta última de CORBA es poder tener una máquina A ejecutando un programa, y que un programa ejecutado desde otra máquina B se pueda utilizar un objeto creado en el programa de la máquina A. Las máquinas A y B pueden ser la misma o estar a kilómetros de distancia.

  Lo interesante de CORBA, es que no está atado a ningún lenguaje; la idea es que los programas de la máquina A y B pueden estar escritos en Java y C, o Perl y Python, o en C++ y Ada. CORBA permitiría que los objetos de cada uno de estos lenguajes se comuniquen entre sí.

  CORBA es un ejemplo de lo que en software se conoce como *middleware*, ya que funciona como intermediario entre aplicaciones construidas en distintos tipos de plataformas.

En esta práctica veremos enchufes, ya que existen en casi todos los lenguajes de programación del universo y porque son relativamente sencillos de utilizar, al precio de no proveer tanta funcionalidad o nivel de abstracción como el RMI, los Servlets/JSP o CORBA.

Actividad 12.2 Consulta la documentación de los paquetes `java.net`, `java.rmi`, y `org.omg.CORBA`.

Enchufes

Los enchufes, contrario a lo que pudiera pensarse, funcionan como enchufes. Piensen en el enchufe telefónico; es un punto de entrada/salida al exterior. Pueden recibir información a través de él (cuando escuchan), y mandar información a través de él (cuando hablan). De hecho, pueden hablar y oír al mismo tiempo; pueden recibir y enviar información al mismo tiempo.

La idea de los enchufes es crear puntos de entrada/salida entre dos computadoras; una vez creados, las computadoras podrán enviarse bytes mutuamente a través de ellos. Es importante señalar que la comunicación se reduce a bytes, por lo que son de muy bajo nivel.

Para establecer la conexión entre dos enchufes, se necesitan dos programas. Se podría hacer la conexión con un solo programa utilizando hilos de ejecución, pero no tiene sentido ya que lo que queremos es comunicar dos máquinas.

El primer programa es llamado *servidor*, y lo que hace es estar *escuchando* en un puerto de comunicación de la máquina donde está corriendo, esperando por una solicitud de conexión. Cuando la recibe crea un enchufe, y si la solicitud es válida, la comunicación queda establecida.

El segundo programa es llamado *cliente*, y lo que hace es crear un enchufe con la dirección de la máquina y el puerto donde está escuchando el servidor. Si el servidor está en esa máquina escuchando en ese puerto, la conexión queda establecida.

Una vez que la conexión ha sido establecida, cada enchufe dispone de un objeto de la clase InputStream y de otro objeto de la clase OutputStream. Cuando un enchufe manda bytes a su OutputStream, el otro los recibe por su InputStream, y viceversa. El control de los enchufes queda totalmente en manos del programador, y de acuerdo a la aplicación se verá cómo cada enchufe controla los mensajes que manda y recibe.

El objeto de la clase InputStream de los enchufes tiene implentado el método available, por lo que siempre podemos saber si hay algo que leer de él. Si el metodo available regresa un entero mayor que cero, entonces hay algo que leer. Si regresa cero, no hay nada que leer.

El servidor

Lo que hace un servidor se resume en las siguientes líneas:

```
try {
    int puerto = 10000;
    ServerSocket servidor = new ServerSocket (puerto);
    Socket cliente = servidor.accept(); // Lo ponemos a escuchar.
    manejaCliente (cliente);
} catch (Exception e) {
    /* Crear un enchufe de servidor y ponerlo a escuchar es
       potencialmente peligroso y puede resultar en que sean
       lanzadas varias excepciones. */
}
```

El servidor se queda detenido en la llamada al método accept, y no sale de ahí hasta que alguna solicitud sea recibida. Cuando se recibe la solicitud, el servidor crea un enchufe que conecta con el enchufe del cliente y ahí termina su función; a partir de ese momento el programa utiliza al enchufe que devuelve accept para comunicarse con el cliente.

Dentro de manejaCliente se establece la manera en que el servidor maneja los mensajes enviados y recibidos. De acuerdo a la aplicación, puede que el servidor se limite a mandar información, o tal vez sólo la reciba. Lo más común, sin embargo, es que haga ambas cosas constantemente.

Si se quiere hacer un servidor para múltiples clientes, se hace algo de este estilo:

```
try {
    int puerto = 10000;
```

```
        ServerSocket servidor = new ServerSocket (puerto);
        while (true) {
            // Nos ponemos a escuchar.
            Socket cliente = servidor.accept();
            // Creamos un hilo de ejecución para que maneje al enchufe.
            MiThread procesoCliente = new MiThread (cliente);
            // Disparamos al hilo de ejecución.
            procesoCliente.start();
        }
    } catch (Exception e) {
    }
```

De esta manera el servidor escucha eternamente por el puerto; cuando una conexión es recibida, dispara un hilo de ejecución que maneja al enchufe de la misma manera que lo haría manejaCliente, y vuelve a esperar por otra conexión. Ésta es la manera en que funcionan casi todos los servidores en la red (HTTP, FTP, TELNET, SSH, etc.)

Si se dan cuenta, un servidor así implementado es un programa orientado a eventos, aunque no tenga interfaz gráfica. Los eventos en este caso son las solicitudes de conexión que recibe el servidor.

Un mismo servidor puede tener un número potencialmente infinito de enchufes conectados a un mismo puerto; sin embargo siempre se limita a un número fijo las conexiones concurrentes posibles.[3]

Actividad 12.3 Consulta la documentación de la clase ServerSocket, en el paquete java.net.

El cliente

El cliente es todavía más sencillo. Para crear el enchufe sólo se necesita la dirección del servidor y el puerto donde está escuchando:

```
    try {
        int puerto = 10000;
        // Puede utilizarse un IP numérico, como "132.248.28.60".
        String direccion = "abulafia.fciencias.unam.mx";
        Socket servidor = new Socket (direccion, puerto);
        manejaServidor (servidor);
    } catch (Exception e) {
        /* Crear un enchufe de cliente también genera varias posibles
           excepciones. */
    }
```

En la creación del enchufe se realiza la conexión con el servidor. La función manejaServidor se encarga de manejar los mensajes enviados y recibidos.

[3]El ancho de banda no es gratuito.

Actividad 12.4 Consulta la documentación de la clase Socket, del paquete java.net.

Analisis de un *chat*

Tienes a tu disposición el código fuente de dos clases: `Servidor.java` y `Cliente.java`. Son el servidor y el cliente de un *chat*.

Un chat es un *espacio virtual* donde varios usuarios se conectan. Pueden mandar mensajes, y los mensajes que envían son vistos por todos los usuarios conectados al chat, incluidos ellos mismos. Son bastante comunes en la red, y han sido objeto de estudios sociológicos y de comportamiento de masas (hay gente que asegura, en público incluso, haber conocido a sus parejas en un chat).

¿Cómo funciona un chat de verdad? Hay un servidor, que es el cuarto del chat. Lo único que hace el servidor es estar escuchando por conexiones. Cada vez que se realiza una conexión, el servidor crea un nuevo hilo de ejecución para manejar al nuevo usuario, y avisa de esto a todos los usuarios conectados.

Cada hilo de ejecución del servidor está escuchando todo el tiempo a ver si su cliente dice algo. Si así es, manda de regreso el mensaje a todos los clientes, incluido el suyo. Esto es importante: *cada hilo de ejecución debe poder comunicarse con los demás.*

El cliente funciona muy similarmente; se conecta al servidor, y se queda esperando por mensajes. Si los recibe, lo único que hace es imprimirlos. Para mandar mensajes el cliente, realmente se necesitaría otro hilo de ejecución; mas de eso se encargará la interfaz gráfica, que como ya sabemos tiene su propio hilo de ejecución.

La clase Servidor a la que tienes acceso es algo inútil: sólo permite una conexión a la vez. Sin embargo te permitirá ver cómo funciona la conexión con enchufes.

Actividad 12.5 Compila las clases Servidor y Cliente. No necesitarás ningún *jarfile* ni utilizar la bandera `-classpath`. Ejecuta el servidor en una máquina con la siguiente línea de comandos:

```
# java Servidor
```

El servidor detectará la dirección de la máquina donde lo estés corriendo, y seleccionará por omisión el puerto 12345. Ahora ejecuta el cliente en *otra* máquina con la siguiente línea de comandos:

```
# java Cliente <direccionDelServidor> 12345
```

Necesitarás pasarle la dirección de la máquina donde esté el servidor. Puedes ejecutar ambos programas en una sola máquina, pero lo interesante es hacerlo en *máquinas distintas*.

Es tarea tuya comprender cómo funcionan ambas clases.

12.4. Ejercicios

1. Basándote (si quieres) en el servidor que sólo recibe una conexión, utiliza hilos de ejecución para que pueda recibir varias conexiones.

12.5. Preguntas

1. Piensa en todos los programas que conoces, ya sea que funcionen en Unix/Linux o en cualquier otro sistema operativo. ¿Crees poder hacerte una idea de cómo están programados? Justifica.

Conclusiones

If anything can go wrong, Fix It! (to hell with Murphy!)

– Anti-Murphy's Laws #1

A pesar de la existencia de estudios que afirman que no es trascendente el que haya o no prácticas de laboratorio durante un primer curso de programación, es convicción mía (y de varios profesores que lo han dado en distintas universidades y carreras) que la existencia de las prácticas y de un laboratorio bien planificado y estructurado juegan un papel determinante en la comprensión del curso, y en su éxito en el sentido de que los alumnos que lo concluyan satisfactoriamente sepan programar.

Mientras que es indudable que los contextos local y nacional influyen en esta afirmación (la mayor parte de estos estudios se han realizado en países como Estados Unidos, donde el porcentaje de estudiantes de licenciatura que cuentan con computadora en casa es superior al de México, por ejemplo), también es resultado de la experiencia empírica y de los resultados obtenidos en las primeras generaciones de la carrera.

Las prácticas de laboratorio, entendidas como parte integral del curso y no sólo como complemento, ayudan a los estudiantes a aterrizar los conocimientos teóricos obtenidos en clase, a familizarizarse con la herramienta de trabajo de nosotros los computólogos, y a saber enfrentarse a la máquina desde una etapa temprana en su carrera. No darla, o darla sólo como un apéndice del curso, sería como enseñar a tocar guitarra a alguien con únicamente tablaturas, partituras y teoría musical. El estudiante tiene que aprender a utilizar su herramienta de trabajo.

Esto, por supuesto, no quiere decir que la teoría no sea importante o que sea secundaria. La parte teórica del curso es fundamental y debe ser tratada con el mismo o con más cuidado que la parte práctica. Pero no se puede sencillamente ignorar a la parte práctica; tiene que ser tomada en cuenta desde la planificación misma del curso y debe estar acoplada cuidadosamente con la parte teórica. Una de las bellezas de las ciencias de la computación es que muchas veces podemos aplicarlas de inmediato al mundo real y ver los resultados: no podemos privar a los alumnos de eso. No debemos privar a los alumnos de eso.

Con esto en cuenta, las prácticas de laboratorio deben entonces ir de la mano con las clases teóricas, estar sincronizadas con ellas. Deben estar bien planificadas para ver en qué orden deben darse los temas y dejarse los ejercicios. Deben considerar el calendario, medir el tiempo que se da para que sean resueltas, y prever en cuáles se podrán atorar los alumnos.

Estas prácticas están pensadas con todo esto en mente, y estructuradas de acuerdo. Se escogió al lenguaje de programación Java pensando en todo el poder que ofrece el lenguaje

y las facilidades que otorga para mostrar ciertos temas. Se previó la necesidad de encapsular en bibliotecas opacas varias herramientas para evitarle al alumno complicaciones innecesarias con la sintaxis o con ciertas características del lenguaje, y se le mostró el interior de estas bibliotecas cuando habían alcanzado el nivel de conocimientes que les permitiera entenderlas. Se eligió cuidadosamente el orden de los temas para evitar siempre que fuera posible las dependencias ciclicas entre ellos, y se utilizaron siempre conceptos relativamente intuitivos cuando no se podían evitar. Se pensó en una infraestructura completa para que el alumno tuviera siempre acceso a documentación, código y ayuda en general, haciéndola disponible en un orden que concordara con las prácticas.

Hay que entender también que las prácticas de laboratorio no son la única parte técnica del curso. Las prácticas están planeadas para realizarse en una o dos semanas, y haciendo que los alumnos implementen una solución específica. El diseño general del problema nunca queda totalmente en sus manos. Es por ello importante que se dejen cuatro o cinco proyectos a lo largo del semestre, en donde sea responsabilidad del alumno el hacer el análisis y diseño del problema. Estos proyectos deben ir aumentando gradualmente en complejidad, y debe darse más tiempo para que los estudiantes puedan hacerse cargo del análisis, diseño e implementación del mismo.

Todo esto fue puesto en práctica durante el semestre 2001-II, durante el cual la profesora Elisa Viso y yo impartimos el curso de Introducción a Ciencias de la Computación I utilizando una primera versión de este manuscrito.

La experiencia obtenida a lo largo del curso y las fortalezas y debilidades del mismo fueron utilizadas para mejorar ciertas partes, quitar algunas, y agregar otras.

El curso tuvo un alto grado de reprobación (más del 50 %), pero creemos (por los resultados en exámenes, prácticas y proyectos), que los alumnos que finalizaron el curso satisfactoriamente lo hicieron con un nivel de conocimientos muy alto, y con una capacidad para aterrizar estos conocimientos en programas mucho mayor que la del promedio de estudiantes que han tomado el curso. Parte de estos resultados se deben a las mismas prácticas; pero es indudable que no se hubieran obtenido si el curso no hubiera contado con un titular de la experiencia, calidad, profesionalismo y entrega de la profesora Viso

El alto nivel de reprobación es preocupante, pero no creemos que sea culpa del curso. El curso es difícil, por supuesto, pero eso es algo *inherente* al mismo. Es difícil teórica y prácticamente, y si está bien diseñado representará un reto para la mayor parte de los alumnos. Y hay que entender que reprobar en sí mismo no es algo *malo*. Etimológicamente "reprobar" significa "que tiene que volver a probar".

Las condiciones en que se da el curso tampoco ayudan. Debería poder esperarse que todos los alumnos de la carrera supieran lo que hacían cuando la escogieron, y que todos hubieran llevado algún curso donde se usaran computadoras en el último año del bachillerato. Es muy difícil enseñar a alguien a programar cuando le da miedo apretar una tecla porque siente que va a estallar la máquina.

Si se buscara únicamente un criterio cuantitativo de eficiencia terminal, lo único que se tendría que hacer es ponerle 10 a todos los alumnos al comienzo del curso. Pero no es eso lo que se busca.

No es tarea de la carrera el ofrecer servicios de orientación vocacional ni garantizar el nivel del bachillerato en la UNAM y el país. Eso es tarea de autoridades superiores.[1]

[1]Lo mismo ocurre con estudiantes que pidieron estudiar Derecho o Medicina y terminan en Ciencias de la Computación; es tarea de otras autoridades el evitar estos casos.

Lo que es tarea de la carrera (y de este curso) es garantizar igualdad de oportunidades para todos los alumnos que ingresan a ella. En el caso particular del curso, todos los alumnos tuvieron la oportunidad de usar los laboratorios de la carrera (cuyo funcionamiento *sí es* responsabilidad de la misma) y recibieron el mismo trato por parte del titular del curso y mío. Y aprobaron 26 de 54.

Una vez dado el curso se hicieron cambios importantes. Las prácticas 3 y 4, que antes eran una sola se convirtieron en dos, y las prácticas 6 (herencia) y 8 (excepciones) fueron reescritas casi completamente, atendiendo quejas de los alumnos respecto a inconsistencias o contradicciones en las mismas. Algunos ejercicios fueron eliminados por excesivamente difíciles o triviales, y algunos más se añadieron. Y el diseño de la base de datos que se utiliza en casi todo el curso sufrió varios cambios importantes, que terminaron en la versión que se presenta.

A pesar de los errores que inevitablemente se tenían que encontrar en una primera versión de estas prácticas, la planeación mostró sus resultados. Las prácticas fueron de invaluable ayuda para que entendieran conceptos teóricos complejos, como son herencia, recursión y encapsulamiento de datos, y les permitieron ver de primera mano y experimentar con ellos. Gracias a las prácticas vieron y dominaron conceptos que hasta ahora habían sido prácticamente descartados de los primeros cursos de programación Orientados a Objetos, como son listas, entrada/salida a disco e interfaces gráficas.

Con estas prácticas, para terminar, los estudiantes fueron capaces al final del curso de entender y utilizar hilos de ejecución y enchufes. Varios alumnos hicieron la práctica opcional, con resultados que van desde aceptables hasta sorprendentes. Ver programación en red en el primer semestre, aunque sea a un nivel tan sencillo, es una innovación de este trabajo.

Un profesor (o ayudante) no puede hacer un análisis de un curso que ha dado sin ser hasta cierto punto subjetivo. Dar clases es de las experiencias más maravillosas que hay en la vida, y por tanto no voy a callar que desde mi punto de vista tuvimos la suerte de tener un grupo con varios alumnos innegablemente brillantes y trabajadores, y que además tienen ese fuego en la mirada al comprender el *sin qua non* de un concepto o al ver que su primer programa compila.[2]

Estas prácticas han sido revisadas y corregidas en varias aproximaciones sucesivas, y creemos que pueden ser utilizadas regularmente en el primer curso de programación de ésta y otras carreras en computación, aunque está principalmente enfocada a la carrera de Ciencias de la Computación en la Facultad de Ciencias. Sostenemos la idea de que este primer curso debe darse con Orientación a Objetos, y que mientras no llegue un mejor reemplazo, Java puede ser utilizado de manera más que satisfactoria como primer lenguaje de programación.

Por supuesto, este trabajo es perfectible, y siempre lo será. Pero la única forma eficiente de perfeccionarlo es seguir dando el curso con él, y retroalimentándose de los alumnos que lo tomen.

[2]Generalmente no corre, pero eso no importa.

Apéndice A
El español en computación

Toda especialidad cuenta con una *jerga* propia, que fuera de dicha especialidad resulta incomprensible, incoherente o sencillamente fuera de lugar. En el caso de las Ciencias de la Computación, nuestra jerga está plagada de anglicismos, traducciones literales, y en algunos tristes casos incluso de españolizaciones sin sentido ("*bacupéate* la base de datos").

En este trabajo se trató de conservar el uso debido del español tanto como fue posible, y de usar los términos en español correspondientes que la mayor parte de los computólogos de habla hispana consideramos correctos. Esto último porque no hay una entidad oficial que decida qué vocablos son correctos y cuáles no en Ciencias de la Computación. *Instanciar*, por ejemplo, no existe en el español de la Real Academia, y probablemente nunca exista fuera de nuestra jerga.[1]

Por supuesto, la traducción en español puede parecer artificial o sin sentido, o simplemente no ser del gusto de todos. Eso es subjetivo, pero lo realmente grave es cuando no se encuentra un equivalente en español para algún término. Sigo sin encontrar una traducción satisfactoria al español para cosas como *buffer* o *hash*.

A lo largo del texto se utilizan términos que, para alguien que no esté acostumbrado a la jerga, resultarán incomprensibles, absurdos e incluso erróneos. Cosas como *instanciar*, *implementar* o *inicializar* no existen fuera de nuestra jerga. Otras tienen un significado que no concuerda al 100% con su definición original, como *compilar* o *evento*.[2]

El hecho de que sean términos *agradables* o no al lector es cuestión de gustos personales. A algunos les parecerán apropiados y a otros no: lo importante es usar el vocablo "correcto" en cada caso ("correcto" entre comillas porque, como ya se dijo no hay una autoridad oficial que discuta y decida la validez de cada una de las expresiones).

Lo que sí representa un problema grave es que al momento de redactar un texto relativamente grande (como éste), resulta que no tenemos sinónimos. Los vocablos en español apenas si son aceptados de común acuerdo entre la gente de habla hispana que se dedica a las Ciencias de la Computación; son términos surgidos del deseo de poder transmitir ideas y resultados de nuestra ciencia utilizando para ello nuestro propio idioma. No ha habido tiempo de buscar o inventar sinónimos.

Entonces resulta inevitable que en este trabajo, cuando se ve instanciación de objetos, en un

[1] Lo cual no es necesariamente malo...

[2] Curiosamente, el vocablo *evento* era antes idéntico dentro y fuera de las Ciencias de la Computación. Pero ahora resulta que un *evento* es un acto o presentación, como un mitin o un concierto.

mismo párrafo aparezcan siete vocablos derivados del término *instanciar*, que además de todo no existe en el español. Esto hace la lectura más difícil, y es considerado una mala práctica en redacción.

Y yo ofrecería disculpas, pero realmente no es mi culpa.

Apéndice B
El resto de las leyes

Program complexity grows until it exceeds the capabilities of the programmer who must maintain it.

– Murphy's Laws of Computer Programming #13

Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited.

– Murphy's Laws of Computer Programming #14

Adding manpower to a late software project makes it later.

– Murphy's Laws of Computer Programming #15

Make it possible for programmers to write programs in English, and you will find that programmers can not write in English.

– Murphy's Laws of Computer Programming #16

The documented interfaces between standard software modules will have undocumented quirks.

– Murphy's Laws of Computer Programming #17

The probability of a hardware failure disappearing is inversely proportional to the distance between the computer and the customer engineer.

– Murphy's Laws of Computer Programming #18

Bibliografía

[1] Mathias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* The MIT Press, 2001.

[2] Lynn Andrea Stein. *Interactive Programming In Java.* Publicación futura por The MIT Press. 2002

[3] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java.* Wiley Computer Publishing, 2000.

[4] Mark Grand. *Patterns in Java Volume 1.* Wiley Computer Publishing, 1998.

[5] Fintan Culwin. *Java: An Object First Approach.* Prentice Hall, 1998.

[6] Ken Arnold, James Gosling, David Holmes. *The Java Programming Language, Third Edition*, Addison-Wesley, 2000.

[7] Mary Campione, Kathy Walrath, Alison Huml. *The Java Tutorial: A Short Course on the Basics*, Addison-Wesley, 2000.

[8] David Flanagan. *Java in a Nutshell*, O'Reilly & Associates, 2001.

Índice alfabético