Introducción a Java™

Autor: Carlos Kavka

Departamento de Informática
Universidad Nacional de San Luis
Ejército de los Andes 950
D5700HHW – San Luis – Argentina

Material de apoyo
utilizado en la materia:
Análisis Comparativo de Lenguajes
Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina, 2005

Resumen

Estas notas tienen el objetivo de introducir el lenguaje Java™. No constituyen de ninguna manera un tratamiento completo del lenguaje Java o su interfase de programación de aplicaciones (API). Los lectores interesados pueden complementar este material consultando los libros referenciados o la documentación Java Software Development Kit (SDK) y API[1].

Este apunte es una traducción adaptada del capítulo 2 de las Lecture Notes del Third Workshop on Distributed Laboratory Instrumentation Systems realizado en Trieste, Italia, del 22 de Noviembre al 17 de Diciembre de 2004.

Luego de una breve introducción a los elementos básicos del lenguaje, temas más avanzados tales como entrada salida, manejo de excepciones y programación concurrente son presentados con muchos ejemplos. Los apéndices contienen el código fuente de los ejemplos principales, y se espera que los lectores puedan experimentar con ellos modificándolos y ejecutándolos.

[1] http://java.sun.com/j2se/1.5.0/docs/api/index.html

Índice

1. Introducción

Java es un lenguaje de programación muy poderoso que ha despertado un enorme interés en los últimos años. Es un lenguaje orientado a objetos, concurrente y de propósito general, con una sintaxis similar a la de C (y C++), pero sin las características que hacen que estos lenguajes sean complejos y poco seguros.

Su principal ventaja radica en que el código compilado es independiente de la arquitectura de la computadora en la que los programas serán ejecutados. Internet ha popularizado el uso de Java, debido a que los programas escritos en este lenguaje pueden ser descargados en forma transparente junto con páginas web y ejecutados en cualquier computadora que disponga de un browser con capacidades de ejecución Java.

Sin embargo, Java no está limitado sólo al desarrollo de aplicaciones web. De hecho, ha sido muy utilizado en otros dominios, incluyendo aplicaciones con microcontroladores.

Una aplicación Java es un programa que puede ser ejecutado sin la necesidad de un web browser. Un applet Java, por otro lado, es un programa diseñado para ser ejecutado exclusivamente en un web browser. En esta introducción a Java se considerarán solamente aplicaciones Java y no applets.

Java fue desarrollado por Sun MicroSystem en 1991, como parte de un proyecto de desarrollo de software para dispositivos electrónicos. La versión actual de Java es la 5.0 (también conocida como JDK 1.5.0). Sun provee la JDK (Java Development Kit) libre de costos a través de su web site[1]. Sin duda, este hecho ha contribuido en gran medida a la popularidad del lenguaje Java.

Estas notas asumen que el lector tiene familiaridad con el lenguaje C. De hecho, usualmente Java se puede aprender más fácilmente que C o C++ dado que la mayor parte de los aspectos complejos de C que son frecuente causa de errores no están presentes en Java.

En estas notas no se cubrirán todos los aspectos de Java, en particular no se presentarán detalles sobre la construcción de applets o interfases gráficas. Todos los ejemplos presentados en estas notas están disponibles para experimentación. De hecho, sirven como un complemento importante, y se sugiere que el lector ejecute y modifique los ejemplos con el objetivo de comprender adecuadamente todos los conceptos presentados.

2. La plataforma Java™

Los programas en Java se compilan en Java byte-codes, una clase de lenguaje de máquina independiente de la arquitectura de las computadoras. El programa compilado se ejecuta utilizando un intérprete denominado Máquina Virtual de Java (JVM). La máquina virtual de Java es una computadora abstracta con su propio conjunto de instrucciones y áreas de memoria. Un programa Java compilado puede entonces ser ejecutado en cualquier sistema operativo que disponga

[1] http://java.sun.com

de una máquina virtual de Java.

La principal ventaja de esta aproximación es, por supuesto, la portabilidad. Es decir, el mismo programa Java ya compilado, se puede ejecutar en cualquier computadora que tenga una máquina virtual de Java. El precio que se paga es una velocidad de ejecución más lenta, debido al uso del intérprete. Algunas mejoras, tales como los compiladores *Just in Time* (JIT), permiten que los programas en Java se ejecuten a velocidades comparables con otros lenguajes.

De esta forma, la compilación a Java byte codes ayuda a lograr el objetivo de *escribir el programa una vez, y ejecutarlo en cualquier arquitectura*.

3. Un primer ejemplo

Esta sección presenta un ejemplo muy simple: la típica aplicación *Hola Mundo*. Una vez compilado el programa, éste imprime el mensaje *Hello World!* como resultado de su ejecución.

```
/**
 * Hello World Application
 * Our first example
 */

public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello World!");  // display output
  }
}
```

El ejemplo, aunque de pequeñas dimensiones, involucra muchos conceptos. En esta sección consideraremos solamentes los aspectos principales. El resto será considerado con más detalles en secciones posteriores.

La aplicación Java consiste de una única clase. Su nombre es `HelloWorld` y debe ser definida en el archivo `HelloWorld.java`. El compilador de Java requiere que el nombre de la clase sea el mismo nombre del archivo, sin la extensión.

La clase define solamente un método llamado `main`, que debe ser definido exactamente como está mostrado en el ejemplo:

```
public static void main(String[] args)
```

El método `main` recibe como argumentos un arreglo de strings, y no retorna nada. Es el punto del programa donde comienza la ejecución.

La clase `System` está definida en la Java API (Application Programming Interface) y se utiliza para proveer acceso a la funcionalidad provista por el sistema. La variable de clase `out` es un miembro de la clase `System` y se utiliza para crear el objeto que permite acceder a la salida estándar. El método `println` se invoca a fin de imprimir el string pasado como argumento en la salida estándar:

```
System.out.println("Hello World!");
```

Existen dos tipos básicos de comentarios para documentar programas, y ambos son ignorados por el compilador. Los comentarios de única linea comienzan con los caracteres `//` y los comentarios de múltiples lineas deben ser definidos entre los caracteres `/*` y `*/` como en C. Existe un tercer tipo de comentario que es utilizados por la herramienta de documentación `javadoc`. Se define entre los caracteres `/**` y `*/`, como en el ejemplo:

```
/**
 * Hello World Application
 * Our first example
 */
```

En estos comentarios se pueden incluir comandos especiales y también código HTML, los que luego son utilizados por la herramienta `javadoc` para generar automáticamente documentación que puede ser visualizada a través de un web browser.

4. Ciclo de desarrollo de aplicaciones en Java

Para desarrollar aplicaciones en Java, se deben realizar los siguientes tres pasos: creación del archivo fuente, compilación y ejecución. El ejemplo más simple se puede mostrar utilizando la JDK de Sun Microsystems:

Creación del archivo fuente : Se puede realizar con cualquier editor de textos. El nombre del archivo debe ser el mismo que el nombre de la clase que se desea definir, con la extensión `.java`. El compilador de Java diferencia minúsculas de mayúsculas, por lo que se debe respetar perfectamente la correspondencia entre el nombre de la clase y el nombre del archivo. Una posibilidad es el uso del editor `emacs` como se muestra en el ejemplo siguiente (nota: # representa el prompt de Unix):

```
# emacs HelloWorld.java
```

Compilación : El compilador de Java se invoca con el comando `javac`, y traduce el código fuente en un archivo que contiene los byte-codes que pueden ser ejecutados por una máquina virtual de Java:

```
# javac HelloWorld.java
```

Este comando crea un archivo con el mismo nombre, pero con la extensión `.class`:


```
# ls
HelloWorld.java
HelloWorld.class
```

Ejecución : El archivo class contiene los byte-codes que pueden ser interpretados por una máquina virtual de Java. Para ejecutar esta aplicación, se debe invocar el programa `java` con el nombre de la clase como argumento (sin extensión):

```
# java HelloWorld
Hello World!
```

La herramienta `javadoc` se puede utilizar para generar automáticamente documentación para la clase, y para todas sus componentes. El siguiente comando crea un conjunto de archivos HTML que describen a la clase `HelloWorld`:

```
# javadoc HelloWorld
```

El principal archivo de salida es `HelloWorld.html` y se muestra en la figura 1.

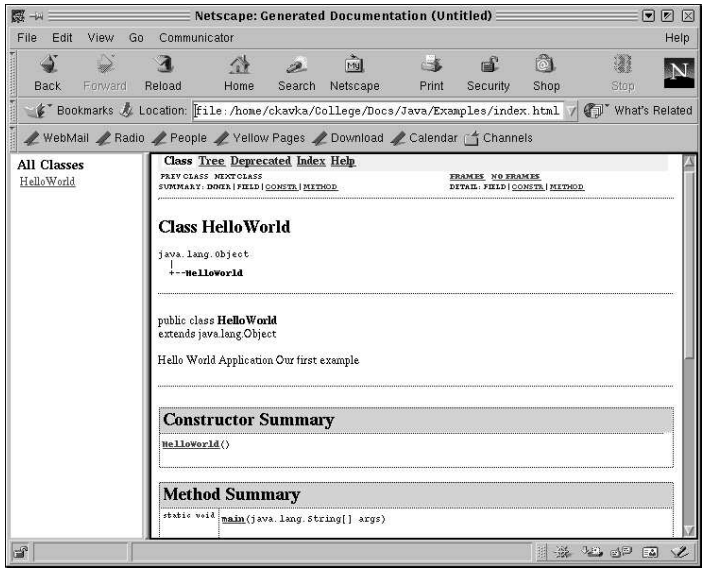

Figura 1: Documentación generada por javadoc

5. Tipos de datos básicos

Java provee diez tipos de datos básicos: cuatro tipos de enteros, dos tipos de números en punto flotante, caracteres, el tipo boolean, el tipo especial `void` y strings. La tabla 1 muestra para algunos tipos de datos, los valores mínimos y máximos que los objetos de estos tipos pueden tomar, junto con su tamaño en bytes.

tipo	tamaño	valor min	valor max
byte	8 bits	-128	127
short	16 bits	-2^{15}	$2^{15}-1$
int	32 bits	-2^{31}	$2^{31}-1$
long	64 bits	-2^{63}	$2^{63}-1$
float	32 bits	$1{,}4\text{E}-45$	$3{,}45\text{E}38$
double	64 bits	$4{,}9\text{E}-324$	$1{,}7\text{E}308$
char	16 bits	unicode 0	unicode $2^{16}-1$

Cuadro 1: Algunas características de los tipos básicos de Java

El tipo `boolean` consiste en dos valores: `true` y `false`, y no tienen equivalencia con valores enteros como ocurre en C.

El tipo `void` se utiliza como tipo de retorno para métodos que no retornan nada, como el método `main` declarado en el primer ejemplo:

```
public static void main(String[] args)
```

El tipo `String` especifica secuencias de caracteres. No está relacionado con arreglos como ocurre en C.

El tipo `char` representa caracteres y es de 16 bits, permitiendo trabajar con el conjunto estándar de caracteres ASCII, más una enorme cantidad de caracteres en distintos lenguajes.

Consideraremos más detalles de los tipos mencionados en secciones siguientes.

6. Variables

Las variables se declaran especificando su tipo y nombre. Pueden ser inicializadas en el punto de su declaración, o se les pueden asignar valores posteriormente a través del operador de asignación, como se muestra en el ejemplo siguiente:

```
int x;                    // not initialized
double f = 0.33;
char c = 'a';
String s = "abcd";

x = 55;                   // value assigned
```


7. Literales

Los valores enteros se pueden escribir en decimal, hexadecimal, octal y en forma larga (long), como se muestra en el siguiente ejemplo:

```
int x = 34;               // decimal value 34
int y = 0x3ef;            // hexadecimal value 3ef
int z = 0772;             // octal value 772
long m = 240395922L;      // long value 240395922
```

Los valores en punto flotante son de tipo `double` por defecto. Para especificar un valor de tipo `float`, se debe agregar la letra `F` al final, como se muestra a continuación:

```
double d = 6.28;          // 6.28 is a double value
float f = 6.28F;          // 6.28F is a float value
```

Los caracteres se especifican con la notación estándar de C, con la excepción de los caracteres en Unicode, que se introducen con `\u`:

```
char c = 'a';             // character lowercase a
char d = '\n';            // newline character
char e = '\u2122'         // unicode character (TM)
```

Los valores de tipo `boolean` son `true` y `false`. Son los dos únicos valores que pueden ser asignados a variables de tipo `boolean`:

```
boolean ready = true;     // boolean value true
boolean late = false;     // boolean value false
```

8. Constantes

La declaración de constantes es similar a la declaración de variables, pero incluyendo la palabra clave `final` al comienzo. La especificación de valor inicial es obligatoria, como se muestra en los ejemplos siguientes:

```
final double pi = 3.1415;          // constant PI
final int maxSize = 100;           // integer constant
final char lastLetter = 'z';       // last lowercase letter
final String word = "Hello";
```

Por supuesto, una vez declaradas las constantes, su valor no puede ser modificado.

9. Expresiones

Java provee un conjunto de operadores muy completo para la construcción de expresiones. Las expresiones se pueden clasificar en: aritméticas, a nivel de bits, relacionales, lógicas y específicas para strings. Se detallan en las siguientes secciones.

9.1. Operadores aritméticos

Java provee el conjunto estándar de operadores aritméticos: suma (+), resta (-), división (/), multiplicación (*) y módulo (%). La siguiente aplicación muestra algunos ejemplos:

```
/**
 * Arithmetic Application
 */
class Arithmetic {

  public static void main(String[] args) {
    int x = 12;
    int y = 2 * x;
    System.out.println(y);
    int z = (y - x) % 5;
    System.out.println(z);
    final float pi = 3.1415F;
    float f = pi / 0.62F;
    System.out.println(f);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
24
2
5.0669355
```

La última parte de esta aplicación muestra que las variables se pueden declarar en cualquier parte del cuerpo de un método. Se pueden utilizar para almacenar un valor desde ese punto hasta el final del bloque en el que fueron definidas.

Java provee operadores de asignación compuestos que están formados por el operador de asignación y un operador binario. Se pueden utilizar para abreviar operaciones de asignación. La siguiente aplicación muestra algunos ejemplos:

```
/**
 * Shorthand operators Application
 */
class ShortHand {

  public static void main(String[] args) {
    int x = 12;
    x += 5;                  // x = x + 5
    System.out.println(x);
    x *= 2;                  // x = x * 2
    System.out.println(x);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
17
34
```

Una operación usual consiste en incrementar o decrementar el valor de una variable. Los operadores $++$ y $--$ realizan respectivamente estas operaciones. Hay dos versiones de ambos operadores, llamadas prefijo y postfijo. En el caso de los operadores de pre-incremento y pre-decremento, la operación se realiza primero, y luego se retorna el valor. En el caso de los operadores de post-incremento y post-decremento, el valor se retorna y luego la operación se efectúa.

La siguiente aplicación presenta algunos ejemplos:

```
/**
 * Increment operator Application
 */

class Increment {

 public static void main(String[] args) {
  int x = 12,y = 12;

  System.out.println(x++); // x is printed and then incremented
  System.out.println(x);

  System.out.println(++y); // y is incremented and then printed
  System.out.println(y);
 }
}
```

La salida producida por la ejecución de la aplicación es:

```
12
13
13
13
```

9.2. Operadores relacionales

Java provee el conjunto estándar de operadores relacionales: equivalencia (==), no equivalencia (!=), menor que (<), mayor que (>), menor o igual que (<=) y mayor o igual que (>=). Las expresiones relacionales siempre retornan un valor de tipo `boolean`.

El siguiente ejemplo muestra el valor retornado por algunas expresiones relacionales.

```
/**
 * Boolean operator Application
 */
class Boolean {
  public static void main(String[] args) {
    int x = 12,y = 33;

    System.out.println(x < y);
    System.out.println(x != y - 21);

    boolean test = x >= 10;
    System.out.println(test);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
true
false
true
```

9.3. Operadores a nivel de bits

Java provee un conjunto de operadores que pueden manipular bits en forma directa. Algunos operadores, tales como *and* (&), *or* (|) y *not* (~) realizan álgebra de Bool sobre bits. Otros realizan desplazamientos: a izquierda (<<), a derecha con extensión de signo (>>) y a derecha sin extensión de signo (>>>).

El operador binario *and* (&) realiza un operación *and* entre los bits de los dos argumentos. El operador binario *or* (|) realiza una operación *or* entre los bits de los dos argumentos. El operador unario *not* (~) realiza una operación *not* sobre los bits de su único argumento.

El operador de desplazamiento a izquierda (<<) desplaza los bits del primer argumento tantas posiciones como indica el segundo argumento, insertando ceros desde el lado correspondiente al bit menos significativo. El operador de desplazamiento a derecha (>>>) desplaza los bits del primer argumento tantas posiciones como indica el segundo argumento insertando ceros desde el lado correspondiente al bit más significativo. El operador de desplazamiento a derecha

con extensión de signo (>>) desplaza los bits del primer argumento tantas posiciones como indica el segundo argumento insertando ceros o unos desde el lado correspondiente al bit más significativo, de forma tal de mantener el signo del primer argumento. Esto significa que se insertan ceros si el número es positivo y unos si el número es negativo.

Estos operadores trabajan con valores enteros. Si el argumento es un `char`, `short` o `byte`, se transforman primero a un `int` y el resultado es de tipo `int`.

El siguiente ejemplo muestra el valor retornado por algunas expresiones con operadores que trabajan a nivel de bits:

```
/**
 * Boolean algebra bit level operators Application
 */

class Bits {

 public static void main(String[] args) {

  int x = 0x16;             // 00000000000000000000000000010110
  int y = 0x33;             // 00000000000000000000000000110011
  System.out.println(x & y);// 00000000000000000000000000010010
  System.out.println(x | y);// 00000000000000000000000000110111
  System.out.println(~x);   // 11111111111111111111111111101001

  x &= 0xf;                 // 00000000000000000000000000000110
  System.out.println(x);    // 00000000000000000000000000000110

  short s = 7;              // 0000000000000111
  System.out.println(~s);   // 11111111111111111111111111111000
 }
}
```

El ejemplo muestra que operadores de asignación compuestos se pueden construir también combinando el operador de asignación y los operadores a nivel de bits binarios (& and |). Los comentarios han sido incorporados en el ejemplo para mostrar los valores obtenidos al ejecutar cada una de las operaciones. Los últimos dos comentarios muestran que si el valor del argumento de un operador *not* es un `short`, el resultado es un `int`.

El siguiente ejemplo muestra los valores retornados por algunas expresiones que involucran desplazamientos a nivel de bits:

```
/**
 * Bit level operators Application
 */
class Bits2 {
```

```
 public static void main(String[] args) {
  int x = 0x16;              //00000000000000000000000000010110
  System.out.println(x << 3);//00000000000000000000000010110000

  int y = 0xfe;              //00000000000000000000000011111110
  y >>= 4;                   //00000000000000000000000000001111
  System.out.println(y);     //00000000000000000000000000001111

  x = 9;                     //00000000000000000000000000001001
  System.out.println(x >> 3);//00000000000000000000000000000001
  System.out.println(x >>>3);//00000000000000000000000000000001

  x = -9;                    //11111111111111111111111111110111
  System.out.println(x >> 3);//11111111111111111111111111111110
  System.out.println(x >>>3);//00011111111111111111111111111110
 }
}
```

9.4. Operadores lógicos

Java provee los operadores lógicos *and* (&&), *or* (||) y *not* (!). Los operadores lógicos se pueden aplicar sólo a expresiones de tipo `boolean` y retornan un valor `boolean`.

El siguiente ejemplo muestra los valores retornados por algunas expresiones lógicas:

```
/**
 * Logical operators Application
 */

class Logical {

  public static void main(String[] args) {
    int x = 12,y = 33;
    double d = 2.45,e = 4.54;

    System.out.println(x < y && d < e);
    System.out.println(!(x < y));

    boolean test = 'a' > 'z';
    System.out.println(test || d - 2.1 > 0);
  }
}
```

La salida producida por la ejecución de la aplicación es:


```
true
false
true
```

9.5. Operadores de strings

Java provee un conjunto muy completo de operadores para strings. La mayoría de ellos serán presentados en secciones posteriores, y sólo se considerará aquí al operador de concatenación (+). Este operador combina dos strings y produce un nuevo string con los caracteres de ambos argumentos.

Cuando una expresión comienza con un string y utiliza a continuación el operador de concatenación, se produce un efecto muy interesante: el siguiente argumento es convertido a `String` si es necesario y luego se realiza la concatenación.

La siguiente aplicación muestra algunos ejemplos:

```
/**
 * Strings operators Application
 */

class Strings {

  public static void main(String[] args) {

    String s1 = "Hello " + "World!";
    System.out.println(s1);

    int i = 35,j = 44;
    System.out.println("The value of i is "  + i +
                       " and the value of j is " + j);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
Hello World!
The value of i is 35 and the value of j is 44
```

Dado que la expresión entre paréntesis comienza con un `String` y se utiliza el operador +, los valores de `i` y `j` se convierten a strings y luego son concatenados:

```
System.out.println("The value of i is "  + i +
                   " and the value of j is " + j);
```

9.6. Casting

Java realiza conversión automática de tipos cuando no hay riesgo de perder información. Es el caso típico de conversiones que *agrandan* el valor (*widening conversions*), tal como muestra el siguiente ejemplo:

```
/**
 * Test Widening conversions Application
 */

class TestWide {

  public static void main(String[] args) {
    int a = 'x';        // 'x' is a character
    long b = 34;        // 34 is an int
    float c = 1002;     // 1002 is an int
    double d = 3.45F;   // 3.45F is a float
  }
}
```

Para poder especificar conversiones donde se puede potencialmente perder información (*narrowing* conversions), es necesario usar el operador de cast. Se especifica indicando entre paréntesis el nombre del tipo al que se desea convertir el resultado de la evaluación de la expresión, tal como muestra el siguiente ejemplo:

```
/**
 * Test Narrowing conversions Application
 */

class TestNarrow {

  public static void main(String[] args) {
    long a = 34;
    int b = (int)a;          // a is a long
    double d = 3.45;
    float f = (float)d;      // d is a double
  }
}
```

Estas conversiones deben ser utilizadas sólo cuando hay certeza absoluta que no se perderá información.

10. Estructuras de control

Java provee el mismo conjunto de estructuras de control que C para la secuencia, selección e iteración. La principal diferencia radica en que las expresiones condicionales son valores de tipo `boolean` y no pueden ser enteros. Las principales estructuras se detallan en las siguientes secciones.

10.1. Sentencia de selección

El mecanismo básico de selección es la sentencia `if`, la cual se utiliza para decidir que sentencia ejecutar en base al valor de una expresión de tipo boolean. Tiene dos formas:

```
if (boolean-expression)
  statement
```

y:

```
if (boolean-expression)
  statement
else
  statement
```

Un *statement* puede ser reemplazado por una única instrucción o por un conjunto de instrucciones encerradas entre llaves.

La siguiente aplicación presenta un ejemplo del uso de la selección `if`, imprimiendo las palabras *letter*, *digit* u *other character* dependiendo del valor de la variable `c`:

```
/**
 * If control statement Application
 */

class If {

  public static void main(String[] args) {

    char c = 'x';

    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
      System.out.println("letter: " + c);
    else if (c >= '0' && c <= '9')
      System.out.println("digit: " + c);
    else {
      System.out.println("other character:");
      System.out.println("the character is: " + c);
      System.out.println("it is not a letter");
      System.out.println("and it is not a digit");
    }
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
letter: x
```

10.2. Sentencias de iteración

Java provee las sentencias de iteraciones `while` y `do-while` tal como C. Permiten la repetición de una sentencia (o sentencia compuesta) mientras el valor de una expresión de tipo `boolean` tiene valor `true`. Sus formas son:

```
while (boolean-expression)
  statement
```

y:

```
do
  statement
while (boolean-expression);
```

Notar que la estructura `while` puede ejecutar la sentencia cero o más veces, mientras que la estructura `do-while` la puede ejecutar una o más veces, dependiendo del valor de la expresión de tipo `boolean`.

El siguiente ejemplo imprime el número de veces que es necesario incrementar una variable en un cierto paso (step) a partir de un valor inicial hasta que alcance un determinado valor límite:

```
/**
 * While control statement Application
 */

class While {

  public static void main(String[] args) {
    final float initialValue = 2.34F;
    final float step = 0.11F;
    final float limit = 4.69F;

    float var = initialValue;
    int counter = 0;

    while (var < limit) {
      var += step;
      counter++;
    }
    System.out.println("It is necessary to increment it "
                       + counter + " times");
  }
}
```


La salida producida por la ejecución de la aplicación es:

```
It is necessary to increment it 22 times
```

Java provee un tercer tipo de iteración al igual que C: la repetición `for`. Su forma es:

```
for(initialization;boolean-expression;step)
  statement;
```

La expresión de inicialización se ejecuta primero, luego la sentencia es repetidamente ejecutada mientras la condición de tipo `boolean` evalúa a `true`. Al final de la ejecución de la sentencia, y antes de la siguiente evaluación de la expresión de tipo `boolean`, la tercera expresión (step) se evalúa.

El siguiente ejemplo realiza los mismos cálculos que el ejemplo anterior, pero usando la iteración `for`:

```
/**
 * For control statement Application
 */

class For {

  public static void main(String[] args) {
    final float initialValue = 2.34F;
    final float step = 0.11F;
    final float limit = 4.69F;

    int counter = 0;

    for (float var = initialValue;var < limit;var += step)
      counter++;

    System.out.println("It is necessary to increment it "
                       + counter + " times");
  }
}
```

El alcance de la variable `var` definida en la primera expresión del `for` es el cuerpo de la iteración, la expresión de tipo `boolean` y la tercera expresión (step). La salida de la aplicación es, por supuesto, la misma que en el ejemplo anterior.

10.3. break y continue

Las sentencias `break` y `continue` proveen una forma de controlar el flujo de ejecución dentro de las iteraciones. `break` termina la iteración mientras que

`continue` comienza una nueva ejecución de la repetición evaluando la expresión de tipo `boolean` nuevamente. En el caso de una repetición `for`, la ejecución de un `continue` hace que la tercera expresión (step) se ejecute antes de evaluar la condición.

El siguiente ejemplo ilustra el uso de `break` y `continue`:

```
/**
 * Break and Continue control statement Application
 */

class BreakContinue {

  public static void main(String[] args) {
    int counter = 0;

    for (counter = 0;counter < 10;counter++) {

      // start a new iteration if the counter is odd
      if (counter % 2 == 1) continue;

      // abandon the loop if the counter is equal to 8
      if (counter == 8) break;

      // print the value
      System.out.println(counter);
    }
    System.out.println("done.");
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
0
2
4
6
done.
```

Notar que la expresión de tipo `boolean` de la primera selección `if` evalúa a verdadero cuando el valor del contador es impar. En este caso, la sentencia `continue` termina anticipadamente la ejecución del cuerpo de la repetición, ejecutando la tercera expresión del `for` (`counter++`), y evaluando la expresión de tipo `boolean` nuevamente (`counter < 10`). Si la expresión evalúa nuevamente a verdadero, el cuerpo de la iteración se ejecuta otra vez.

La sentencia `break` termina la ejecución completa de la iteración cuando el contador alcanza el valor 8, causando que el control de ejecución se transfiera a la última sentencia del programa.

Aunque `break` y `continue` se pueden utilizar también combinados con rótulos, no cubriremos este uso en estas notas.

10.4. Sentencia de control Switch

La estructura de control `switch` selecciona bloques de códigos para su ejecución en base al valor de una expresión entera. Su estructura es la siguiente:

```
switch (integral-expression) {

  case integral-value: statement; [break;]
  ...
  case integral-value: statement; [break;]

  [default: statement;]
}
```

Los corchetes encierran sentencias opcionales. La sentencia cuyo valor entero asociado coincide con el valor de la evaluación de la expresión entera es seleccionada para su ejecución. Si se utiliza la sentencia `break` opcional, la ejecución del `switch` completo se termina, en caso contrario, todas las sentencias que siguen son ejecutadas independientemente de su valor entero asociado, hasta que se encuentre un `break`, o se alcance el fin del cuerpo del `switch`.

La expresión entera puede ser cualquier expresión que retorne un valor que se pueda convertir a `int`. Esto significa que puede ser de tipo `char`, `short`, `byte` o `int`.

El siguiente ejemplo cuenta el número de días en un año. Notar que la respuesta será diferente si las sentencias `break` son eliminadas.

```
/**
 * Switch control statement Application
 */

class Switch {

  public static void main(String[] args) {
    boolean leapYear = true;
    int days = 0;

    for(int month = 1;month <= 12;month++) {

      switch(month) {
        case 1:         // months with 31 days
        case 3:
```

```
        case 5:
        case 7:
        case 8:
        case 10:

        case 12:
          days += 31;
          break;

        case 2:         // February is a special case
          if (leapYear)
            days += 29;
          else
            days += 28;
          break;

        default:        // it must be a month with 30 days
          days += 30;
          break;
      }
    }
    System.out.println("number of days: " + days);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
number of days: 366
```

11. Arreglos

En Java se pueden declarar arreglos para almacenar elementos de un mismo tipo. A continuación se presentan algunos ejemplos de declaraciones de arreglos:

```
int[] a;        // an unitialized array of integers
float[] b;      // an unitialized array of floats
String[] c;     // an unitialized array of Strings
```

Las declaraciones no especifican el tamaño de los arreglos, incluso, no asignan espacio para los elementos. El tamaño se puede especificar inicializando los arreglos en la misma declaración:

```
int[] a = {13,56,2034,4,55};            // size: 5
float[] b = {1.23F,2.1F};               // size: 2
String[] c = {"Java","is","great"};     // size: 3
```


Otra posibilidad consiste en asignar espacio para los arreglos utilizando el operador `new`. En este caso, el tamaño de los arreglos se puede incluso calcular en tiempo de ejecución:

```
int i = 3,j = 5;
double[] d;           // unitialized array of doubles

d = new double[i+j];  // array of 8 doubles
```

Cuando se utiliza el operador `new` la memoria se asigna en forma dinámica. Las componentes del arreglo se inicializan con los valores por defecto: `0` para elementos de tipo numérico, `'\0'` para caracteres y `null` para referencias (en secciones posteriores se presentarán más detalles).

Las componentes de los arreglos se pueden acceder usando un índice entero que toma valores entre 0 y el tamaño del arreglo menos 1. Por ejemplo, se puede modificar el tercer elemento (aquel que tiene índice 2) del arreglo `a` del primer ejemplo, con la siguiente asignación:

```
a[2] = 1000;    // modify the third element of a
```

Todos los arreglos tienen un campo (o dato miembro) llamado `length` que puede ser utilizado para obtener la longitud (número de elementos). La siguiente aplicación muestra algunos ejemplos del uso de arreglos:

```
/**
 * Arrays Application
 */

class Arrays {

  public static void main(String[] args) {
    int[] a = {2,4,3,1};

    // compute the summation of the elements of a
    int sum = 0;
    for(int i = 0;i < a.length;i++)
      sum += a[i];

    // create an array of floats with this size
    float[] d = new float[sum];

    // assign some values
    for(int i = 0;i < d.length;i++)
      d[i] = 1.0F / (i + 1);

    // print the values in odd positions
```

```
    for(int i = 1;i < d.length;i += 2)
      System.out.println("d[" + i + "]=" + d[i]);

  }
}
```

La salida producida por la ejecución de la aplicación es:

```
d[1]=0.5
d[3]=0.25
d[5]=0.16666667
d[7]=0.125
d[9]=0.1
```

Se pueden declarar también arreglos multidimensionales utilizando una aproximación similar. Por ejemplo, la siguiente linea de código declara una matriz de enteros que puede ser utilizada para almacenar 50 elementos, organizados en 10 filas de 5 columnas.

```
int[][] a = new int[10][5];
```

12. Argumentos en línea de comandos

Hemos visto en ejemplos anteriores que el método `main` debe ser definido como sigue:

```
public static void main(String[] args)
```

El método `main` toma un único argumento que está definido como un arreglo de strings. A través de este arreglo, el programa puede acceder a los argumentos pasados cuando la aplicación es invocada para su ejecución utilizando la máquina virtual de Java. La siguiente aplicación imprime todos los argumentos pasados en la línea de comando:

```
/**
 * Command Line Arguments Application
 */

class CommandArguments {

  public static void main(String[] args) {

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


A continuación se presentan algunas ejecuciones de esta aplicación:

```
# java CommandArguments Hello World
Hello
World
# java CommandArguments
# java CommandArguments I have 25 cents
I
have
25
cents
```

Notar que en el último ejemplo el argumento 25 es un entero, sin embargo es considerado por la máquina virtual de Java como el string "25", el cual se almacena en `args[2]` y consta de los caracteres `'2'` y `'5'`. Se puede convertir un string en un entero válido usando el método `parseInt` que pertenece a la clase `Integer` (en secciones siguientes se presentarán más detalles).

La siguiente aplicación acepta dos argumentos en la línea de comandos, que deben ser enteros, e imprime el resultado de su suma:

```
/**
 * Add Application
 */

class Add {

  public static void main(String[] args) {

    if (args.length != 2) {
      System.out.println("Error");
      System.exit(0);
    }
    int arg1 = Integer.parseInt(args[0]);
    int arg2 = Integer.parseInt(args[1]);

    System.out.println(arg1 + arg2);
  }
}
```

Algunas ejecuciones se presentan a continuación:

```
# java Add 2 4
6
# java Add 4
Error
# java Add 33 22
55
```

Notar el uso del método `exit` que pertenece a la clase `System`: su objetivo es terminar la ejecución de la aplicación.

13. Clases

Una clase se define en Java usando la palabra clave `class` y especificando a continuación un nombre para ella. Por ejemplo, el código:

```
class Book {

}
```

declara una clase llamada `Book`. Se pueden crear instancias de la clase usando la palabra clave `new`, como muestran los siguientes ejemplos:

```
Book b1 = new Book();
Book b2 = new Book();
```

o en dos pasos, con exactamente el mismo significado:

```
Book b3;

b3 = new Book();
```

Por supuesto, esta clase no es muy útil dado que su cuerpo está vacío (es decir, no contiene nada).

Dentro de una clase se pueden definir datos miembros, usualmente denominados *campos*, y funciones, usualmente denominadas *métodos*. Los campos son utilizados para almacenar información y los métodos se utilizan para la comunicación con las instancias de la clase.

Con el objeto de utilizar las instancias de la clase `Book` para almacenar información sobre libros, en particular el título, el autor, y el número de páginas de cada libro, se pueden agregar tres campos a la definición de la clase `Book` tal como se muestra a continuación:

```
class Book {
  String title;
  String author;
  int numberOfPages;
}
```

Ahora, cada instancia de esta clase contiene tres campos. Los campos pueden ser accedidos utilizando el operador punto, el que consiste en el uso de un punto (.) entre el nombre de la instancia y el nombre del campo que se desea acceder.

La siguiente aplicación muestra como crear una instancia y como acceder a estos campos:


```
/**
 * Example with books Application
 */

class Book {
  String title;
  String author;
  int numberOfPages;
}

class ExampleBooks {

  public static void main(String[] args) {
    Book b;

    b = new Book();                // default constructor
    b.title = "Thinking in Java";
    b.author = "Bruce Eckel";
    b.numberOfPages = 1129;

    System.out.println(b.title + " : " + b.author +
                       " : " + b.numberOfPages);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
Thinking in Java : Bruce Eckel : 1129
```

13.1. Constructores

Los constructores permiten la creación de instancias inicializadas adecuadamente. Un constructor es un método que tiene el mismo nombre que la clase a la cual pertenece, y no tiene especificación del tipo de retorno.

La siguiente aplicación define un constructor llamado `Book` (no hay otra opción) que inicializa todos los campos de una instancia de `Book` con los valores pasados como argumentos:

```
/**
 * Example with books Application (version 2)
 */

class Book {
  String title;
  String author;
  int numberOfPages;
```

```
  Book(String tit,String aut,int num) {      // constructor
    title = tit;
    author = aut;
    numberOfPages = num;
  }
}

class ExampleBooks2 {

  public static void main(String[] args) {
    Book b;

    //  create an instance of a book

    b = new Book("Thinking in Java","Bruce Eckel",1129);


    System.out.println(b.title + " : " + b.author +
                       " : " + b.numberOfPages);
  }
}
```

El constructor se invoca cuando una instancia de la clase `Book` se crea. La salida producida por la ejecución de la aplicación es:

```
Thinking in Java : Bruce Eckel : 1129
```

Java provee un constructor por defecto para todas las clases. Este constructor fue invocado en el ejemplo `ExampleBooks` mostrado previamente, sin argumentos:

```
b = new Book();
```

El constructor por defecto está disponible sólo cuando no se han definido otros constructores para la clase. Esto significa que en el último ejemplo (`ExampleBooks2`) no se pueden crear instancias de la clase `Book` utilizando el constructor por defecto.

Es posible definir más de un constructor para una misma clase, siempre y cuando tengan distinto número de argumentos o distintos tipos para ellos. De esta forma, el compilador será capaz de identificar cual es el constructor que debe ser llamado cada vez que una instancia es creada.

La siguiente aplicación agrega un campo adicional a los libros: el número ISBN. El constructor definido previamente se modifica de forma tal de asignar un valor adecuado a este campo. Se agrega también un nuevo constructor para inicializar todos los campos con los valores pasados como argumentos. Notar que el compilador no tiene problemas en identificar el constructor que se desea invocar cuando se crean instancias:


```
/**
 * Example with books Application (version 3)
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }
}

class ExampleBooks3 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    System.out.println(b1.title + " : " + b1.author +
        " : " + b1.numberOfPages + " : " + b1.ISBN);


    b2 = new Book("Thinking in Java","Bruce Eckel",1129,
        "0-13-027363-5");
    System.out.println(b2.title + " : " + b2.author +
        " : " + b2.numberOfPages + " : " + b2.ISBN);
  }
}
```

La salida de la ejecución de la aplicación es:

```
Thinking in Java : Bruce Eckel : 1129 : unknown
Thinking in Java : Bruce Eckel : 1129 : 0-13-027362-5
```

13.2. Métodos

Los métodos se utilizan para implementar los mensajes a los que una instancia (o clase) puede responder. Se implementan como funciones, especificando los argumentos y el tipo de valor de retorno. Se invocan utilizando la notación punto.

La siguiente aplicación es igual a la definida anteriormente, pero incluye un método que permite obtener las iniciales del nombre del autor a partir de una instancia de la clase `Book`:

```
/**
 * Example with books Application (version 4)
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }


  public String getInitials() {
    String initials = "";

    for(int i = 0;i < author.length();i ++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
      }
    }
    return initials;
  }
```



```
}

class ExampleBooks4 {

  public static void main(String[] args) {
    Book b;

    b = new Book("Thinking in Java","Bruce Eckel",1129);
    System.out.println("Initials: " + b.getInitials());
  }
}
```

La salida de la ejecución de la aplicación es:

```
Initials: B.E.
```

El prototipo del método `getInitials()` es:

```
public String getInitials()
```

El método se define con el modificador `public` de forma que pueda ser llamado desde otras clases (se darán más detalles en secciones posteriores), no toma argumentos y retorna un string. Se puede invocar utilizando la notación punto:

```
System.out.println("Initials: " + b.getInitials());
```

Notar que no se pasan argumentos en la invocación. En terminología de la programación orientada a objetos, se dice que el mensaje `getInitials` se envía al objeto `b`, el que se denomina receptor del mensaje.

El método se implementa como sigue:

```
  public String getInitials() {
    String initials = "";


    for(int i = 0;i < author.length();i ++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
      }
    }
    return initials;
  }
```

Todas las referencias al campo `author` corresponden al campo denominado con ese nombre en el receptor del mensaje, en este caso, de la instancia `b`.

El método crea un string vacío en la variable `initials`, y recorre el campo `author` buscando letras mayúsculas. Todas las letras mayúsculas son concatenadas en el string `initials` seguidas por un punto.

Los métodos `length()` y `charAt(int)` de la clase `String` son utilizados respectivamente para obtener la longitud de un string y el caracter en la posición especificada en el string.

El siguiente ejemplo define un arreglo de instancias de la clase `Book`, inicializándolas con datos de tres libros. Luego, se invoca el método `getInitials` sobre las tres instancias. Debería ser ahora claro que aún si el método `getInitials` procesa los datos almacenados en el campo `author`, corresponde en cada caso a los campos así denominados en los distintos receptores del mensaje.

```
class ExampleBooks5 {

  public static void main(String[] args) {
    Book[] a;

    a = new Book[3];
    a[0] = new Book("Thinking in Java","Bruce Eckel",1129);
    a[1] = new Book("Java in a nutshell","David Flanagan",353);
    a[2] = new Book("Java network programming",
                    "Elliotte Rusty Harold",649);

    for(int i = 0;i < a.length;i++)
      System.out.println("Initials: " + a[i].getInitials());
  }
}
```

La salida de la ejecución de la aplicación es:

```
Initials: B.E.
Initials: D.F.
Initials: E.R.H.
```

13.3. Igualdad y equivalencia

El operador usual para verificar la igualdad (==) puede tener un comportamiento confuso cuando se utiliza para comparar objetos. La siguiente aplicación define dos libros con los mismos valores y luego los compara:

```
class ExampleBooks6 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
```



```
    b2 = new Book("Thinking in Java","Bruce Eckel",1129);

    if (b1 == b2)
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

La salida de la ejecución de la aplicación es:

```
The two books are different
```

Este comportamiento se debe a que el operador de equivalencia (==) analiza si los dos objetos pasados como argumentos son el mismo objeto, sin tener en cuenta si sus campos tienen o no los mismos valores. El comportamiento es diferente, por supuesto, si la aplicación se modifica como sigue:

```
class ExampleBooks6a {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = b1;

    if (b1 == b2)
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

La salida de la ejecución de la aplicación es:

```
The two books are the same
```

En esta aplicación, `b1` y `b2` son referencias al mismo objeto. La expresión `b1 == b2` retorna `true`, debido a que ambas variables hacen referencia exactamente al mismo objeto.

Para poder verificar la igualdad de dos objetos teniendo en cuenta los valores de los campos, es necesario definir un método específico. Es común que este método se llame `equals`. Se puede definir en la clase `Book` tal como lo muestra el siguiente ejemplo:

```
  public boolean equals(Book b) {
    return (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }
```

El método `equals` recibe una referencia a una instancia de la clase `Book` como un argumento y retorna un valor de tipo `boolean`. Este valor se calcula a través de una expresión que compara cada campo en forma individual.

La siguiente aplicación verifica la igualdad de libros:

```
class ExampleBooks7 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Thinking in Java","Bruce Eckel",1129);

    if (b1.equals(b2))
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

La salida de la ejecución de la aplicación es:

```
The two books are the same
```

13.4. Campos estáticos

En la programación orientada a objetos, las clases se utilizan como modelos para crear instancias. Cada instancia de la clase `Book` tiene cuatro campos (`title`, `author`, `numberOfPages` e `ISBN`), los que permiten almacenar valores en cada instancia en forma completamente independiente de las otras que puedan haber sido creadas. Por esta razón se denominan variables de instancia.

Los campos estáticos (o variables de clases) son campos que pertenecen a la clase y no tienen existencia individual en cada instancia. Significa que siempre existe una única copia de este campo, independientemente del número de instancias que puedan haber sido creadas.

El siguiente ejemplo define un campo estático llamado `owner`, el que será utilizado para almacenar el nombre del dueño de los libros, asumiendo que todos los libros que se definirán en la aplicación pertenecerán a la misma persona. En este caso, claramente, no es necesario tener un campo individual en cada instancia para almacenar el nombre del dueño, dado que su valor sería el mismo

en todas las instancias. El ejemplo siguiente define dos métodos: `setOwner` y `getOwner`, los que serán utilizados para almacenar y obtener el dueño de todos los libros:

```
/**
 * Example with books Application (version 8)
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;
  static String owner;

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }

  public String getInitials() {
    String initials = "";

    for(int i = 0;i < author.length();i ++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
      }
    }
    return initials;
  }

  public boolean equals(Book b) {
    return (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }
```

```
  public void setOwner(String name) {
    owner = name;
  }

  public String getOwner() {
    return owner;
  }
}

class ExampleBooks8 {

  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Java in a nutshell","David Flanagan",353);
    b1.setOwner("Carlos Kavka");

    System.out.println("Owner of book b1: " + b1.getOwner());
    System.out.println("Owner of book b2: " + b2.getOwner());
  }
}
```

La aplicación crea dos libros, y almacena el nombre del dueño enviando el mensaje `setOwner` al objeto `b1` (también podría haberse utilizado `b2`). Luego imprime el nombre del dueño de ambos libros. La salida de la ejecución de la aplicación es:

```
Carlos Kavka
Carlos Kavka
```

Se puede apreciar que aún si el nombre fue almacenado enviando un mensaje al objeto `b1`, el dueño de `b2` fue modificado. De hecho, tal como fue explicado anteriormente, existe un único campo `owner`, el que puede ser accedido con el método `getOwner` utilizando cualquier instancia de la clase `Book`.

Los campos estáticos se utilizan para la comunicación entre las distintas instancias de una clase, o para almacenar valores globales a nivel de clases.

13.5. Métodos estáticos

Siguiendo la misma idea de los campos estáticos, se pueden definir métodos estáticos, los que en general se denominan métodos de clase. Estos métodos no operan con las instancias, sino directamente con la clase. Por ejemplo, suponga que se desea definir un método denominado `description` para proveer información sobre la clase `Book`. Esta información es general, y el valor retornado

por el método debe ser la misma, independientemente de cualquier instancia. El método se puede definir como sigue:

```
  public static String description() {
    return "Book instances can store information on books";
  }
```

Notar el uso de la palabra clave `static` antes de la especificación del valor de retorno. La aplicación puede llamar al método como sigue:

```
class ExampleBooks9 {

  public static void main(String[] args) {

    Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);

    System.out.println(b1.description());
    System.out.println(Book.description());
  }
}
```

La salida de la ejecución de la aplicación es:

```
Book instances can store information on books
Book instances can store information on books
```

Los métodos estáticos se llaman enviando el mensaje a la clase o a cualquier instancia. Es importante tener en cuenta que en la definición de métodos estáticos, sólo se pueden acceder a campos estáticos, dado que su valor no depende de las instancias particulares. En el ejemplo anterior, el único campo que se puede acceder desde el método `description` es `owner`.

13.6. Una aplicación estática

Los ejemplos vistos hasta el momento definen un clase que contiene un método estático llamado `main`, donde en general se han creado instancias de otras clases. Es posible también definir una clase donde todos los campos y los métodos sean estáticos, tal como muestra el siguiente ejemplo:

```
/**
 * All static class Application
 */

import java.io.*;

class AllStatic {
  static int x;
```

```
  static String s;

  public static String asString(int aNumber) {
    return "" + aNumber;
  }

  public static void main(String[] args) {

    x = 165;
    s = asString(x);
    System.out.println(s);
  }
}
```

Esta aplicación define dos campos estáticos cuyos nombres son `x` y `s`, y dos métodos estáticos con nombres `asString` y `main`. El método `main` llama al método `asString`, lo que es perfectamente válido, dado que ambos son estáticos y ambos operan sólo sobre campos estáticos. No hay necesidad de crear una instancia de esta clase para comenzar a enviar mensajes.

En cierto sentido, cuando se utilizan sólo campos y métodos estáticos, la clase se parece a un programa en C, con funciones y datos globales.

Notar la forma en la que el método `asString` convierte un valor entero en un string: utiliza el operador + y la propiedad que tiene este operador de concatenación de convertir el segundo argumento a un string.

13.7. Inicialización de campos

Todos los campos de un objeto tiene un valor inicial adecuado. Existe un valor por defecto para cada tipo primitivo tal como muestra la tabla 2:

tipo	valor por defecto
byte	0
short	0
int	0
long	0
float	0.0F
double	0.0
char	'\0'
boolean	false

Cuadro 2: Valores iniciales para los tipos primitivos

Todas las referencias a objetos tienen el valor inicial `null`. La siguiente aplicación muestra un ejemplo:


```
/**
 * InitialValues Application
 */

class Values {
  int x;
  float f;
  String s;
  Book b;
}

class InitialValues {

  public static void main(String[] args) {

    Values v = new Values();

    System.out.println(v.x);
    System.out.println(v.f);
    System.out.println(v.s);
    System.out.println(v.b);
  }
}
```

La salida de la ejecución de la aplicación es:

```
0
0.0
null
null
```

Los valores pueden ser inicializados también en el constructor, o incluso llamando métodos en el punto de la declaración, como muestra el siguiente ejemplo:

```
/**
 * InitialValues Application (version 2)
 */

class Values {
  int x = 2;
  float f = inverse(x);
  String s;
  Book b;

  Values(String str) {
```

```
    s = str;
  }

  public float inverse(int value) {
    return 1.0F / value;
  }
}

class InitialValues2 {

  public static void main(String[] args) {

    Values v = new Values("hello");

    System.out.println(v.x);
    System.out.println(v.f);
    System.out.println(v.s);
    System.out.println(v.b);
  }
}
```

La salida de la ejecución de la aplicación es:

```
2
0.5
hello
null
```

14. La palabra clave this

La palabra clave `this`, cuando se usa en la definición de un método, hace referencia al objeto receptor. Tiene dos usos principales: 1) se puede utilizar para retornar una referencia al objeto receptor del método y 2) se puede utilizar para llamar constructores desde otros constructores.

Por ejemplo, el método `setOwner` definido en la clase `Book` podría haber sido definido de la siguiente manera:

```
  public Book setOwner(String name) {
    owner = name;
    return this;
  }
```

El valor de retorno del método se define como una referencia a `Book`. El valor retornado es una referencia al objeto receptor a través del uso de `this`. Con esta definición, el método se puede invocar como sigue:


```
Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
System.out.println(b1.setOwner("Carlos Kavka").getInitials());
System.out.println(b1.getOwner());
```

El mensaje `setOwner` se envía a `b1` y este método retorna una referencia al objeto receptor, el cual es el mismo `b1`. Luego el mensaje `getInitials` se envía a `b1`.

La salida de la ejecución de la aplicación es:

```
B.E.
Carlos Kavka
```

En este ejemplo, `this` fue utilizado para retornar una referencia al objeto receptor. El otro uso posible de `this` es para invocar un constructor a partir de otro constructor. Por ejemplo, en la definición de la clase `Book` fueron definidos dos constructores:

```
  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }
```

El segundo constructor se puede definir en una forma más compacta llamando al primer constructor como muestra el siguiente ejemplo:

```
  Book(String tit,String aut,int num,String isbn) {
    this(tit,aut,num);
    ISBN = isbn;
  }
```

El efecto es exactamente el mismo: el primer constructor se invoca a través de `this`, y luego el valor del parámetro `isbn` se asigna al campo `ISBN`.

Cuando la palabra `this` se utiliza con este sentido, debe ser la primera acción realizada por el constructor.

15. Un ejemplo: la clase de los números complejos

En esta sección se presenta una clase que permite trabajar con números complejos. La siguiente aplicación muestra un ejemplo del uso de la clase `Complex`, la que será definida posteriormente.

```
/**
 * Test Complex class Application
 */

class TestComplex {

  public static void main(String[] args) {

    Complex a = new Complex(1.33,4.64);
    Complex b = new Complex(3.18,2.74);

    Complex c = a.add(b);
    System.out.println("a+b = " + c.getReal() + " " +
                       c.getImaginary());

    Complex d = c.sub(a);
    System.out.println("c-a = " + d.getReal() + " " +
                       d.getImaginary());

  }
}
```

Esta aplicación crea dos números complejos `a` y `b` con valores iniciales para sus partes reales e imaginarias. Un número complejo `c` se crea a continuación con el valor correspondiente a la suma de los números `a` y `b`, y luego los valores correspondientes a sus partes real e imaginaria son impresos. Posteriormente, se crea el número complejo `d` a partir de la diferencia entre `c` y `a`, valor que también es impreso en la salida estándar.

La salida de la ejecución de la aplicación es como sigue:

```
a+b = 4.51 7.38
c-a = 3.18 2.74
```

La clase `Complex` debería tener dos campos para almacenar la parte real y la parte imaginaria de los números complejos. Es necesario también definir un constructor que inicialice ambas partes a partir de argumentos, y además métodos para obtener las partes reales e imaginarias. Esto se puede hacer como se muestra a continuación:

```
/**
```



```
 * Complex Number class
 */

public class Complex {
  double real;         // real part
  double im;           // imaginary part

  /** This constructor creates a complex number from its real
   *  and imaginary part.
   */
  Complex(double r,double i) {
    real = r;
    im = i;
  }

  /** This method returns the real part
   */

  public double getReal() {
    return real;
  }

  /** This method returns the imaginary part
   */

  public double getImaginary() {
    return im;
  }
}
```

Se deben también definir dos métodos específicos para implementar la suma y la diferencia de números complejos. A partir del ejemplo, se puede ver que ambos métodos deben tomar un único argumento: el número complejo que debe ser sumado o restado del número que es receptor del mensaje. Por ejemplo, en la siguiente expresión, el método `sub` debe restar el número `a` del número complejo `c`:

```
Complex d = c.sub(a);
```

Notar que los métodos tienen que crear un nuevo número complejo y retornarlo como resultado. No deben modificar el receptor ni el número que fue pasado como argumento. Estos métodos se pueden implementar de la siguiente manera:

```
  /** This method returns a new complex number which is
   *  the result of the addition of the receptor and the
```

```
   *  complex number passed as argument
   */

  public Complex add(Complex c) {
    return new Complex(real + c.real,im + c.im);
  }

  /** This method returns a new complex number wich is
   *  the result of the substraction of the receptor and the
   *  complex number passed as argument
   */

  public Complex sub(Complex c) {
    return new Complex(real - c.real,im - c.im);
  }
```

Notar que se utiliza el método `new` para crear una nueva instancia de un número complejo y su inicialización se realiza invocando al constructor. Luego, el valor es retornado.

Para analizar una forma distinta de definición de métodos, considere la definición de un método `addReal` cuyo objetivo es incrementar solamente la parte real del receptor en un valor que es pasado como argumento. Notar que este método, a diferencia de `add` y `sub`, debe modificar el receptor del mensaje. Un ejemplo de su uso es:

```
a.addReal(2.0);
```

Considerando el ejemplo previo, se deberían obtener los valores 3.33 y 4.64 en las partes real e imaginaria respectivamente de `a` después de la ejecución del método. Otro uso puede ejemplificarse de la siguiente manera:

```
a.addReal(2.0).addReal(3.23);
```

En este caso, el efecto consiste en sumar primeramente 2.0 a la parte real de `a`, y luego el valor 3.23.

Para lograr este efecto, es necesario que el método `addReal` retorne una referencia al objeto receptor, de forma tal que la siguiente invocación a `addReal` pueda operar sobre el mismo número complejo. Dado que `this` en el cuerpo de un método hace referencia al objeto receptor, se puede implementar como sigue:

```
  /** This method increments the real part by a value
   *  passed as argument. Note that the method modifies
   *  the receptor
   */

  public Complex addReal(double c) {
```



```
    real += c;
    return this;
  }
```

Se debe tener cuidado cuando se desea crear un nuevo número complejo como una copia de otro número, dado que una expresión de asignación, tal como la que se muestra abajo, puede no tener el efecto esperado:

```
Complex e = a;
```

Esta asignación hace que `e` sea una referencia al mismo objeto referenciado por `a` (ver sección 13.3). Esto significa que, por ejemplo, si incrementamos `e`, el número complejo `a` también será incrementado.

A fin de crear un nuevo número complejo, se debe utilizar un constructor como se muestra a continuación:

```
Complex e = new Complex(a);
```

Para esto es necesario entonces definir un nuevo constructor que tome un número complejo como argumento. Una forma interesante de hacerlo es la siguiente:

```
  /** This constructor creates a complex number as a copy
   *  of the complex number passed as argument
   */

  Complex(Complex c) {
      this(c.real,c.im);
  }
```

Notar que este constructor toma un número complejo como argumento, e invoca (usando `this`) al constructor definido anteriormente.

La implementación completa de la clase `Complex`, incluyendo también otros métodos adicionales, se presenta en el apéndice B.

16. Herencia

La herencia permite la definición de nuevas clases reusando clases definidas previamente. Se puede definir una nueva clase (denominada subclase) especificando que debe ser *como* otra clase (denominada super clase) utilizando en la definición la palabra clave `extends` seguida del nombre de la super clase. La definición de la nueva clase especifica solamente las diferencias con la super clase.

Supongamos que se desea extender la definición de la clase `Book` definida en la sección 13 con el objeto de almacenar información sobre libros científicos. La

nueva clase contendrá dos campos más que los provistos por la clase `Book` que almacenarán el área de la ciencia sobre la que tratan y un campo de tipo `boolean` que permitirá distinguir proceedings (anales) de libros científicos comunes:

```
class ScientificBook extends Book {
  String area;
  boolean proceeding = false;
}
```

Las instancias de un `ScientificBook` contienen seis campos cada una: `title`, `author`, `numberOfPages`, `ISBN`, `area` y `proceeding`; los cuatro campos heredados de la super clase `Book`, y los dos campos que han sido definidos en esta clase. Notar que con esta definición, un libro científico no es por defecto un proceeding.

16.1. Constructores

Es posible definir un constructor para la clase como sigue:

```
ScientificBook(String tit,String aut,int num,String isbn,
               String a) {
  super(tit,aut,num,isbn);
  area = a;
}
```

Este constructor tiene los mismos parámetros que uno de los constructores de la clase `Book`, y un parámetro adicional que representa el área. Debido a que existe un constructor en la clase `Book` que permite inicializar sus cuatro primeros campos, no es necesario hacerlo nuevamente aquí. Se puede llamar al constructor de la super clase utilizando la palabra clave `super`.

Utilizando este constructor, es posible crear una instancia de esta clase como se muestra a continuación:

```
ScientificBook sb;

sb = new ScientificBook("Neural Networks, A Comprehensive
        Foundation","Simon Haykin",696,"0-02-352761-7",
        "Artificial Intelligence");
```

El método `super` debe ser la primera instrucción en el cuerpo del constructor. Si no es así, el compilador de Java inserta una llamada a `super` sin parámetros. Si no existe un constructor con este formato, el compilador indica un error.

Es posible definir una jerarquía completa de clases a través de herencia. Cada clase puede heredar de las otras que están definidas por encima, y agregar nuevos campos y métodos.

16.2. Métodos

Las clases pueden tener dos o incluso más métodos con el mismo nombre, pero con conjuntos distintos de parámetros. Java es capaz de distinguir los métodos en base al número, tipo y orden de los parámetros. Esta propiedad se conoce como *sobrecarga de métodos*.

En una subclase se pueden definir nuevos métodos para especificar el comportamiento de los objetos de esa clase. Sin embargo, como es esperable, también es posible invocar directamente los métodos definidos en clases que están ubicadas más arriba en la jerarquía.

Los nuevos métodos que se definen en una subclase pueden tener exactamente el mismo nombre y los mismos parámetros que métodos definidos en super clases. Esta propiedad se conoce como *ocultamiento de métodos*.

Cuando se envía un mensaje a un objeto, se busca el método correspondiente en la clase del objeto receptor. Si no es encontrado, la búsqueda prosigue sucesivamente en niveles más altos de la jerarquía hasta que es encontrado.

Uno de los objetivos de la herencia consiste en reusar código definido en otras clases. En algunos casos, sin embargo, el comportamiento de un método debe ser cambiado en la subclase. Este efecto se logra redefiniendo al método. Ya que la búsqueda de los métodos comienza siempre a partir de la clase del objeto receptor, el método más específico siempre es seleccionado.

Considerando nuevamente el ejemplo de los libros, es posible reutilizar el método `getInitials` de la clase `ScientificBook`, dado que trabaja solamente sobre el campo `author`, que es común a instancias de ambas clases. Sin redefinirlo en la clase de los libros científicos, se puede utilizar como muestra el siguiente ejemplo:

```
System.out.println(sb.getInitials());
```

donde `sb` es la instancia de la clase `ScientificBook` que fue definida anteriormente.

Notar que no se puede utilizar el método `equals` para comparar instancias de libros científicos, dado que para determinar la igualdad es necesario comparar dos campos adicionales. Es posible, sin embargo, reusar el método `equals` de la clase `Book` y agregar solamente la comparación para los nuevos campos de la siguiente manera:

```
public boolean equals(ScientificBook b) {
  return super.equals(b) && area.equals(b.area ) &&
      proceeding == b.proceeding;
}
```

El método `equals` compara los campos `area` y `proceeding`. La comparación de los otros cuatro campos se realiza invocando el método `equals` definido en la super clase, utilizando `super`. De esta forma, esta definición del método `equals` redefine el método con el mismo nombre definido en la super clase. Sin embargo, el método `equals` de la super clase es utilizado como parte de la definición del

nuevo método. El uso de `super` para llamar a métodos de la super clase puede ocurrir en cualquier parte del cuerpo de la definición de los métodos.

Debería quedar claro que el método podría alternativamente haber sido definido tal como se muestra a continuación, dado que todos los campos son accesibles:

```
public boolean equals(ScientificBook b) {
  return (title.equals(b.title) && author.equals(b.author) &&
          numberOfPages == b.numberOfPages &&
          ISBN.equals(b.ISBN) && area.equals(b.area) &&
          proceeding == b.proceeding;
}
```

Por supuesto, la versión anterior permite reusar código, algo que no ocurre en esta definición.

No es necesario invocar al método redefinido en la subclase. Por ejemplo, el método `description` podría ser definido para retornar un valor en forma completamente independiente del método con el mismo nombre en la super clase:

```
public static String description() {
  return "ScientificBook instances can store information" +
         " on scientific books";
}
```

También se pueden definir métodos nuevos. Por ejemplo, se pueden definir métodos para manipular el campo `proceeding` tal como se muestra a continuación:

```
public void setProceeding() {
  proceeding = true;
}

public boolean isProceeding() {
  return proceeding;
}
```

Notar que es posible enviar el mensaje `setProceeding` a una instancia de la clase `ScientificBook`, pero no es posible enviarlo a instancias de la la clase `Book`.

La siguiente aplicación presenta un ejemplo del uso de libros científicos:

```
/**
 * Test Scientific Book Class
 */

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



```
   ScientificBook sb1,sb2;

   sb1 = new ScientificBook("Neural Networks, A Comprehensive"+
             " Foundation","Simon Haykin",696,"0-02-352761-7",
             "Artificial Intelligence");
   sb2 = new ScientificBook("Neural Networks, A Comprehensive"+
             " Foundation","Simon Haykin",696,"0-02-352761-7",
             "Artificial Intelligence");

   sb2.setProceeding();

   System.out.println(sb1.getInitials());
   System.out.println(sb1.equals(sb2));
   System.out.println(sb2.description());
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
S.H.
false
ScientificBook instances can store information on
scientific books
```

La definición completa de la clase `ScientificBook` se presenta en el apéndice C.

16.3. Métodos instanceof y getClass

El método `instanceof` retorna un valor de tipo `boolean` que indica si un objeto es una instancia de una clase específica. El método `getClass` retorna la clase de un objeto, la cual puede (entre otras cosas) ser impresa como un string. Como un ejemplo, la siguiente aplicación muestra un resultado interesante en el contexto de herencia:

```
/**
 * Test Class Application
 */

class TestClass {

  public static void main(String[] args) {
   Book b1;
   ScientificBook sb1;
```

```
   b1 = new Book("Thinking in Java","Bruce Eckel",1129);

   sb1 = new ScientificBook("Neural Networks, A Comprehensive"+
             " Foundation","Simon Haykin",696,"0-02-352761-7",
             "Artificial Intelligence");

   System.out.println(b1.getClass());
   System.out.println(sb1.getClass());
   System.out.println(b1 instanceof Book);
   System.out.println(sb1 instanceof Book);
   System.out.println(b1 instanceof ScientificBook);
   System.out.println(sb1 instanceof ScientificBook);
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
class Book
class ScientificBook
true
true
false
true
```

Las dos invocaciones a `getClass` retornan las clases de las que cada uno de los objetos receptores son instancias, cuyos nombres son impresos a continuación. Cuando se invoca a `instanceof`, se puede apreciar que `b1` es una instancia de la clase `Book` y no una instancia de la clase `ScientificBook`, mientras que `sb1` es una instancia de `ScientificBook`. Es importante notar que `sb1` es también una instancia de la clase `Book`. De hecho, todos los objetos son instancias de su propia clase, y de todas las clases que están en niveles superiores de la jerarquía de clases. Esto es lo que permite que las instancias de `ScientifBook` puedan aceptar mensajes que corresponden a métodos definidos en la clase `Book`.

17. Paquetes

Un paquete (package) es una estructura que permite organizar un conjunto de clases. No hay restricciones con respecto al número de clases que un paquete puede contener. Normalmente se agrupan en un paquete a clases que tienen un mismo objetivo o que están relacionadas por herencia.

Las clases básicas provistas por Java están organizadas en paquetes. Por ejemplo, Java provee la clase `Date` para trabajar con fechas en nuestras aplicaciones.. Esta clase está definida en el paquete `java.util`. Se utiliza la palabra clave `import` para indicar al compilador de Java los nombres de las clases que

se desean utilizar en las aplicaciones. Por ejemplo, para indicar que se desea utilizar la clase `Date` definida en el paquete `java.util`, se debe incluir la siguiente línea:

```
import java.util.Date;
```

También se puede especificar sólo el nombre del paquete y no el nombre de la clase, para indicarle al compilador que hay interés en utilizar alguna (o todas) las clases definidas en un paquete:

```
import java.util.*;
```

Como un ejemplo, la siguiente aplicación importa la definición de la clase `Date` para imprimir la fecha actual:

```
/**
 * Test Date Class
 */

import java.util.*;

class TestDate {

  public static void main(String[] args) {

    System.out.println(new Date());
  }
}
```

La salida de la ejecución de la aplicación (en el momento en el que se ejecutó) es:

```
Wed Sep 15 11:40:16 ART 2004
```

Se pueden definir nuevos paquetes utilizando la palabra clave `package` con el nombre del paquete que se desea definir como argumento. Por ejemplo:

```
package mypackage;
```

hace que la clase que se está por definir se integre al paquete `mypackage`. Esta instrucción debe ser la primer sentencia del archivo (no se consideran los comentarios). Las clases definidas en el archivo pertenecerán al paquete `mypackage`. También puede haber otros archivos que definan clases a ser incluidas en el mismo paquete. Luego podrán ser importadas desde otras clases utilizando la sentencia `import`.

Existe una convención con respecto a los nombres de los paquetes para que puedan ser compartidos fácilmente en la comunidad de desarrolladores Java. Siguiendo esta convención, el nombre de los paquetes se define utilizando la dirección internet de los desarrolladores, la que se refleja en la estructura de directorios utilizada para almacenar los archivos de las clases. Se sugiere que el lector consulte las referencias para más información.

18. Control de accesos

A través de los denominados *modificadores de acceso* se puede especificar en un clase cual es el nivel permitido de acceso a los métodos y campos desde las otras clases. Los tres modificadores son `public`, `private` y `protected`. Existe también el nivel de acceso por defecto, que permite acceso completo desde todas las clases que pertenecen a un mismo paquete. Este es el nivel de acceso que fue utilizado en todos los ejemplos hasta ahora. Acceso completo significa que es posible acceder a todos los campos y métodos de una clase desde las otras clases. Por ejemplo, se puede especificar que un libro científico es un proceeding desde la clase `TestScientificBook` como muestra el siguiente ejemplo:

```
sb1.setProceeding();
```

Esto también se puede realizar de la siguiente forma:

```
sb1.proceeding = true;
```

En general no se debería permitir acceso directo a un campo desde fuera de la clase, de forma tal de garantizar una encapsulación adecuada. Para forzar la encapsulación, se puede utilizar el modificador `private`. Este modificador garantiza que el campo puede ser accedido solamente desde los otros métodos que pertenecen a la clase, y no desde fuera de la clase. Por ejemplo, la clase `ScientificBook` se puede definir como sigue:

```
class ScientificBook extends Book {
  private String area;
  private boolean proceeding = false;
}
```

Entonces, el acceso directo al campo `proceeding` no se permite desde las otras clases, y la condición de que un libro científico sea proceeding puede ser establecida solamente enviando el mensaje `setProceeding`.

El mismo concepto se aplica a la definición de métodos. Un método privado puede ser invocado solamente desde los otros métodos de su misma clase.

En general, los campos se definen como privados, y solamente pueden ser modificados por los métodos que pertenecen a su misma clase. Esto es de hecho una propiedad muy importante de los tipos de datos abstractos (ADT) denominada encapsulación.

El modificador `public` permite acceso completo desde todas las otras clases sin restricciones. Es la forma usual en la que los métodos se definen, de forma tal que los mensajes que ellos implementan puedan ser enviados a objetos de esa clase desde cualquier otra clase.

El modificador `protected` permite el acceso a los campos y métodos desde las subclases y desde todas las demás clases en el mismo paquete.

19. final y abstract

Los dos modificadores adicionales que también pueden ser utilizados en la definición de métodos y clases son: `final` y `abstract`.

Un método final no puede ser redefinido en una subclase, lo que significa que no es posible cambiar el comportamiento que él especifica en las subclases. Un método abstracto no tiene cuerpo, y debe ser redefinido en las subclases. Esto significa que es posible definir clases que fuercen a las subclases a definir un método especificado.

Una clase abstracta es una clase que no puede ser instanciada, o en otros términos, no es posible crear instancias de clases abstractas. Sin embargo, como es posible definir subclases, se pueden crear instancias de subclases de clases abstractas.

A continuación se presenta un ejemplo que involucra todos estos conceptos. Suponga que se desea definir una aplicación que utiliza diferentes tipos de placas de entrada salida. En particular, suponga que se desea trabajar con placas de comunicación serial y ethernet. Se deben definir entonces dos clases, una para cada tipo de placa.

Sin embargo, existen datos que son comunes a todo tipo de placa de entrada salida: nombre, contador de errores, etc. También existen operaciones que son comunes: inicialización, lectura, escritura, etc.

Una buena decisión de diseño consiste en definir una clase denominada `IOBoard` que contenga los campos y los métodos que son comunes a todos los tipos de placa de entrada salida. Luego, se pueden definir subclases que implementen cada uno de los tipos particulares de placa.

La clase `IOBoard` debe ser abstracta, con el sentido que no sea posible definir instancias de esta clase general de placa de entrada salida, pero que permita crear instancias de sus subclases.

Es importante notar que no es posible definir el código que realice la comunicación real con los dispositivos en esta clase abstracta `IOBoard` dado que por su caracter general no puede asumir un tipo específico de hardware. Esto fuerza a que el código que incluye los detalles de implementación de la comunicación que es dependiente de la placa particular sea definido en las subclases. Los métodos entonces son definidos como abstractos, de forma tal de forzar a las subclases a definir sus propias versiones de estos métodos para implementar el comportamiento requerido en cada caso.

El método utilizado para incrementar el contador de errores en la clase abstracta `IOBoard` se pude definir como `final`, dado que ninguna subclase debería cambiar su comportamiento.

A continuación se presenta el código de la clase `IOBoard` definido de acuerdo a lo discutido en los párrafos anteriores:

```
/**
 * IO board Class
 */
```

```
abstract class IOBoard {
  String name;
  int numErrors = 0;

  IOBoard(String s) {
    System.out.println("IOBoard constructor");
    name = s;
  }

  final public void anotherError() {
    numErrors++;
  }

  final public int getNumErrors() {
    return numErrors;
  }
  abstract public void initialize();
  abstract public void read();
  abstract public void write();
  abstract public void close();
}
```

Notar que las subclases de `IOBoard` no pueden redefinir `anotherError` dado que fue declarado como método `final`. Además no es posible crear instancias de la clase `IOBoard` dado que fue declarada abstracta. Esto significa que no es posible hacer algo como:

```
IOBoard b = new IOBoard("my board");  // wrong !!!!
```

La subclase que define la implementación de las placas seriales se puede definir como se muestra a continuación:

```
/**
 * IO serial board Class
 */

class IOSerialBoard extends IOBoard {
  int port;

  IOSerialBoard(String s,int p) {
    super(s);
    port = p;
    System.out.println("IOSerialBoard constructor");
  }

  public void initialize() {
```



```
    System.out.println("initialize method in IOSerialBoard");
    // specific code to initialize a serial board
  }

  public void read() {
    System.out.println("read method in IOSerialBoard");
    // specific code to read from a serial board
  }

  public void write() {
    System.out.println("write method in IOSerialBoard");
    // specific code to write to a serial board
  }

  public void close() {
    System.out.println("close method in IOSerialBoard");
    // specific code to close a serial board

  }
}
```

Esta clase define un constructor que toma el nombre de la placa y un valor entero que corresponde al puerto serial como argumentos. El puerto corresponde al campo definido en esta clase, y el nombre es el valor que es almacenado en el campo definido en la super clase. Notar que el constructor llama al constructor de la super clase utilizando `super`. En este ejemplo, los métodos sólo imprimen un mensaje para identificar su ejecución y retornan, dado que para mantener simple el ejemplo no se incluye código real de manejo de comunicaciones.

La subclase `IOEthernetBoard` se puede definir como se muestra a continuación:

```
/**
 * IOEthernetBoard Class
 */


class IOEthernetBoard extends IOBoard {
  long networkAddress;

  IOEthernetBoard(String s,long netAdd) {
    super(s);
    networkAddress = netAdd;
    System.out.println("IOEthernetBoard constructor");
  }
```

```
  public void initialize() {
    System.out.println("initialize method in IOEthernetBoard");
    // specific code to initialize an ethernet board
  }

  public void read() {
    System.out.println("read method in IOEthernetBoard");
    // specific code to read from an ethernet board
  }

  public void write() {
    System.out.println("write method in IOEthernetBoard");
    // specific code to write to an ethernet board
  }

  public void close() {
    System.out.println("close method in IOEthernetBoard");
    // specific code to close an ethernet board

  }
}
```

Esta clase define un constructor que toma el nombre de la placa y una dirección de red como argumentos. La dirección de red corresponde a un campo definido en esta clase y el nombre al valor que debe ser almacenado en el campo correspondiente de la super clase. Notar que aquí también, el constructor llama al constructor de la super clase utilizando `super`.

La siguiente aplicación muestra un ejemplo de uso de estas clases:

```
/**
 * Test Boards1 class Application
 */
class TestBoards1 {
  public static void main(String[] args) {

    IOSerialBoard serial = new IOSerialBoard("my first port",
                                             0x2f8);

    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

La salida producida por la ejecución de la aplicación es:


```
IOBoard constructor
IOSerialBoard constructor
initialize method in IOSerialBoard
read method in IOSerialBoard
close method in IOSerialBoard
```

Notar el orden en que los constructores son ejecutados y el hecho que los métodos definidos en las subclases son ejecutados y no los definidos en la super clase .

20. Polimorfismo

El polimorfismo es una característica muy importante que define la forma en la que los métodos apropiados para ser ejecutados sobre un objeto son seleccionados entre el conjunto de todos los métodos que tienen un mismo nombre. Se dice que existe polimorfismo cuando diferentes objetos pueden responder a la misma clase de mensajes. De esta forma, se puede operar con estos objetos usando la misma interfase.

En el último ejemplo, se puede apreciar que las instancias de las clases `IOSerialBoard` y `IOEthernetBoard` pueden responder al mismo conjunto de mensajes. En este caso existe entonces polimorfismo. Esta propiedad es la que permite trabajar con instancias de ambas clases de la misma forma, tal como muestra el ejemplo siguiente:

```
/**
 * Test Boards2 class Application
 */

class TestBoards2 {

  public static void main(String[] args) {

    IOBoard[] board = new IOBoard[3];

    board[0] = new IOSerialBoard("my first port",0x2f8);
    board[1] = new IOEthernetBoard("my second port",0x3ef8dda8);
    board[2] = new IOEthernetBoard("my third port",0x3ef8dda9);

    for(int i = 0;i < 3;i++)
      board[i].initialize();

    for(int i = 0;i < 3;i++)
      board[i].read();

    for(int i = 0;i < 3;i++)
```

```
      board[i].close();
  }
}
```

En esta aplicación, se define un arreglo de tres instancias de la clase `IOBoard`. Un problema que en un principio parece existir es que no es posible definir instancias de esta clase, dado que fue declarada abstracta. Sin embargo, tal como fue mencionado anteriormente, las instancias de las subclases de `IOBoard` son también instancias de `IOBoard` (ver sección 16.3). Entonces, es posible hacer asignaciones como las que se muestran en el ejemplo, asignando una instancia de `IOSerialBoard` y dos instancias de `IOEthernetBoard` al arreglo.

Para realizar lectura de datos, las placas deben ser inicializadas, leídas y luego cerrada la comunicación. Dado que la interfase de los distintos tipos de placas es la misma, se puede operar con las instancias almacenadas en el arreglo enviando simplemente los mensajes, sin importar el tipo de placa. Esto es posible debido al polimorfismo.

21. Interfases

El último ejemplo de la sección anterior define una clase abstracta con el objetivo de incluir todos los campos y los métodos comunes que toda placa de entrada salida debe tener e implementar. Es posible extender esta idea con el uso de interfases. Una interfase es similar a una definición de clase, pero todos los campos son `static` y `final`, y los métodos no tienen cuerpo y tienen modificador de acceso `public`. No se pueden crear instancias de interfaces.

Los campos representan entonces valores constantes, y los métodos pueden definir un comportamiento, aunque no lo implementan, tal como su nombre lo indica. Solamente especifican una interfase para su invocación.

Luego es posible definir clases que implementan un interfase determinada utilizando la palabra clave `implements`, lo que fuerza a que la clase implemente todos los métodos definidos en la interfase.

En el ejemplo anterior, si se desea que todas las clases que definen placas de entrada salida tengan un campo para nombre y otro para contador de errores, se podría definir una interfase `IOboardInterface` definida tal como se muestra a continuación:

```
/**
 * IO board interface
 */

interface IOBoardInterface {

  public void initialize();
  public void read();
  public void write();
```



```
  public void close();
}
```

La clase `IOSerialBoard` puede entonces ser definida implementando esta interfase, y no como subclase de otra clase:

```
/**
 * IO serial board Class (second version)
 */

class IOSerialBoard2 implements IOBoardInterface {
  int port;

  IOSerialBoard2(int p) {
    port = p;
    System.out.println("IOSerialBoard constructor");
  }

  public void initialize() {
    System.out.println("initialize method in IOSerialBoard");
    // specific code to initialize a serial board
  }

  public void read() {
    System.out.println("read method in IOSerialBoard");
    // specific code to read from a serial board
  }

  public void write() {
    System.out.println("write method in IOSerialBoard");
    // specific code to write to a serial board
  }

  public void close() {
    System.out.println("close method in IOSerialBoard");
    // specific code to close a serial board

  }
}
```

La siguiente aplicación muestra un ejemplo del uso de esta clase:

```
/**
 * Test Boards3 class Application
 */
```

```
class TestBoards3 {

  public static void main(String[] args) {

    IOSerialBoard2 serial = new IOSerialBoard2(0x2f8);

    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

Las clases pueden implementar más de una interfase. Por ejemplo, la siguiente interfase denominada `niceBehaviour` define un comportamiento que se espera que todas las clases *educadas* implementen:

```
/**
 * Nice behavior interface
 */

interface NiceBehavior {

  public String getName();
  public String getGreeting();
  public void sayGoodBye();
}
```

Para forzar que la clase que define las placas seriales implemente todos los métodos de `IOBoardInterface` y todos los métodos de `NiceBehavior`, se debe definir la clase como sigue:

```
/**
 * IO serial board Class (third version)
 */

class IOSerialBoard3 implements IOBoardInterface,
                                NiceBehavior {
  ...
}
```

El compilador de Java acepta esta definición de clase, sólo si todos los métodos definidos en ambas interfaces están definidos en la clase.

El efecto es similar al que se obtiene con la definición de clases abstractas. En cierto sentido, ambas formas de trabajar fuerzan a las clases a definir un comportamiento específico. Sin embargo, una clase puede implementar más de

una interfase, pero una subclase no puede heredar de más de una super clase. Las interfases permiten implementar una clase de herencia múltiple. Como regla de diseño, la primera solución es preferible y una clase debe ser definida como abstracta cuando hay datos y/o métodos que deben ser definidos en la clase, y compartidos por todas las subclases. Cuando solamente se desea especificar una interfase y no una relación de herencia, la segunda solución es preferible.

22. Excepciones

El comportamiento normal de una aplicación cuando hay un error en tiempo de ejecución consiste en abortar la ejecución. Por ejemplo:

```
/**
 * Test Exceptions class Application
 */

class TestExceptions1 {

  public static void main(String[] args) {

    String s = "Hello";

    System.out.print(s.charAt(10));
  }
}
```

Dado que el string no tiene un caracter en la posición 10, la ejecución se detiene con el siguiente mensaje:

```
Exception in thread "main"
    java.lang.StringIndexOutOfBoundsException:
      String index out of range: 10
        at java.lang.String.charAt(String.java:499)
        at TestExceptions1.main(TestExceptions1.java:11)
```

Este error, o excepción en la terminología de Java, puede ser tomado y procesado usando la sentencia `try catch` como muestra el siguiente ejemplo:

```
/**
 * Test Exceptions class Application (version 2)
 */
class TestExceptions2 {
  public static void main(String[] args) {

    String s = "Hello";
```

```
    try {
      System.out.println(s.charAt(10));
    } catch (Exception e) {
      System.out.println("No such position");
    }
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
No such position
```

Cuando la excepción ocurre dentro del bloque definido por `try`, el control se transfiere al bloque definido por `catch`. Este bloque procesa todo tipo de excepciones. El ejemplo siguiente muestra como se puede especificar que sólo las excepciones de índice fuera de rango en strings deban ser procesadas:

```
/**
 * Test Exceptions class Application (version 3)
 */

class TestExceptions3 {

  public static void main(String[] args) {

    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
    }
  }
}
```

Existe una serie de mensajes que pueden ser enviados a un objeto de tipo excepción. Por ejemplo, la siguiente aplicación envía el mensaje `toString` al objeto excepción `e`:

```
/**
 * Test Exceptions class Application (version 4)
 */

class TestExceptions4 {

  public static void main(String[] args) {
```



```
    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
      System.out.println(e.toString());
    }
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
No such position
java.lang.StringIndexOutOfBoundsException:
   String index out of range: 10
```

Existe un conjunto predefinido de excepciones que se pueden considerar en los programas en Java. Es posible también definir nuevos tipos de excepciones, algo que no será tratado en estas notas. En algunos casos, como se mostrará más adelante, es obligatorio definir bloques que traten con excepciones. También es posible indicar que no hay interés en tratar determinado tipo excepciones. Las siguientes secciones presentan múltiples ejemplos.

23. Entrada Salida

La forma en la que se realiza la entrada salida en Java es relativamente compleja. Hay un gran número de clases que se utilizan para leer y escribir datos. La gran ventaja de la entrada salida en Java es que las operaciones de lectura y escritura en archivos, dispositivos, memoria o sitios web se realiza exactamente de la misma forma.

La entrada salida de Java está implementada en el paquete `java.io`. Está basada en la idea de canales de comunicación (*streams*). Un stream de entrada es una fuente de datos que se puede acceder con el objetivo de obtener datos. Un stream de salida es un canal sobre el que se pueden escribir datos.

Los streams están divididos en *byte streams* y *character streams*. Los streams orientados a bytes se utilizan para leer o escribir datos en unidades pequeñas, tal como bytes, enteros, etc. Los streams orientados a caracteres se usan para escribir y leer caracteres.

Java también permite escribir y leer objetos completos (propiedad conocida como serialización), pero no se cubrirá el tema en estas notas. Las siguientes secciones introducen la forma en la que se trabaja con streams.

23.1. Streams orientados a bytes

Existen dos clases provistas por Java para trabajar con streams orientados a bytes: la clase `FileOutputStream` que se utiliza para escribir bytes en un stream, y la clase `FileInputStream` que se utiliza para leer bytes de un stream.

Como ejemplo, la siguiente aplicación escribe 5 bytes en un archivo con nombre `file1.data`:

```
/**
 * Write bytes class Application
 */
import java.io.*;

class WriteBytes {

  public static void main(String[] args) {

    int data[] = { 10,20,30,40,255 };

    FileOutputStream f;

    try {
      f = new FileOutputStream("file1.data");

      for(int i = 0;i < data.length;i++)
        f.write(data[i]);

      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

La aplicación define una referencia a un `FileOutputStream`. Luego se crea una instancia de esta clase con la palabra clave `new` pasando el nombre de archivo como argumento. El efecto producido por esta operación consiste en relacionar el objeto interno `f` con el archivo `file1.data`, de forma tal, que cuando se realice una operación de escritura sobre `f`, los datos son escritos en el archivo.

En este ejemplo, todos los valores almacenados en las componentes del arreglo son escritos en el archivo utilizando el mensaje `write`. El archivo es finalmente cerrado utilizando el mensaje `close`.

Notar que todas las operaciones con el stream están incluidas en un bloque `try catch`. El bloque es obligatorio, y el compilador produce un mensaje de error si no se incluye, debido a que se puede generar la excepción `IOException` cuando se opera con el stream.

El ejemplo que se presenta a continuación lee los datos del archivo `file1.data`:


```
/**
 * Read bytes class Application
 */
import java.io.*;

class ReadBytes {

  public static void main(String[] args) {

    FileInputStream f;

    try {
      f = new FileInputStream("file1.data");

      int data;
      while((data = f.read()) != -1)
        System.out.println(data);

      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

En este ejemplo se crea una instancia de un `FileInputStream` relacionando el archivo `file1.data` con la variable interna. Los bytes se leen uno a continuación del otro con el mensaje `read`. Este mensaje retorna el byte leído, o `-1` cuando se alcanza el fin del archivo. El stream también se cierra con el mensaje `close`. Notar que en este ejemplo, también es necesario contemplar el manejo de la excepción `IOException`.

La salida de la ejecución de la aplicación es:

```
10
20
30
40
255
```

Java provee también un mensaje `write` para almacenar un arreglo de bytes en forma completa en un archivo. El siguiente ejemplo, similar a la definición de la clase `WriteBytes`, escribe todos los componentes de un arreglo de bytes al mismo tiempo:

```
/**
 * Write bytes class Application
```

```
 */
import java.io.*;

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

    byte data[] = { 10,20,30,40,50 };
    FileOutputStream f;

    try {
      f = new FileOutputStream("file1.data");

      f.write(data,0,data.length);

      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

El mensaje `write` recibe como argumentos el arreglo de bytes, el índice de la primera componente que debe ser almacenada, y el número de ellas. Aún cuando el arreglo sea almacenado de esta forma, puede ser leído sin problemas utilizando el ejemplo previo `ReadBytes`. Existe también un mensaje equivalente que permite leer un arreglo de bytes en una única operación.

Es muy importante notar que el mensaje `write` espera un entero como argumento, y que el método `read` devuelve un entero en lugar de bytes como podría esperarse. Esto se debe a que un byte normal puede tomar valores entre -128 y 127, y el byte escrito en un archivo debe estar en el rango 0 a 255. Sin embargo, los métodos que escriben y leen arreglos completos de bytes trabajan con bytes y no con enteros.

23.2. Streams orientados a bytes con buffer

Con el objeto de minimizar la comunicación entre la aplicación y los dispositivos (o el sistema operativo), es muy común la utilización de buffers. En Java se pueden definir streams con buffers utilizando las clases `BufferedOutputStream` y `BufferedInputStream`. Es necesario definir los streams como fue discutido en la sección anterior, y luego agregar los buffers. Los mensajes disponibles son los mismos.

La siguiente aplicación presenta ejemplos sobre la escritura de archivos utilizando streams con buffers:

```
/**
 * Write buffered bytes class Application
```



```
 */
import java.io.*;

class WriteBufferedBytes {

  public static void main(String[] args) {

    int data[] = { 10,20,30,40,255 };
    FileOutputStream f;
    BufferedOutputStream bf;

    try {
      f = new FileOutputStream("file1.data");
      bf = new BufferedOutputStream(f);

      for(int i = 0;i < data.length;i++)
        bf.write(data[i]);

      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

Una instancia `bf` de stream de salida orientado a bytes con buffer se crea pasando el objeto stream `f` como argumento del constructor. De esta forma, el programador expresa su interés en utilizar buffer para el procesamiento del stream `f`. El mismo concepto se aplica para la lectura:

```
/**
 * Read buffered bytes class Application
 */
import java.io.*;

class ReadBufferedBytes {

  public static void main(String[] args) {

    FileInputStream f;
    BufferedInputStream bf;

    try {
      f = new FileInputStream("file1.data");
      bf = new BufferedInputStream(f);
```

```
      int data;
      while((data = f.read()) != -1)
        System.out.println(data);

      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
10
20
30
40
255
```

23.3. Streams orientados a bytes con buffer para datos

Un stream orientado a bytes con buffer para datos puede ser utilizado para trabajar con datos correspondientes a tipos primitivos. Los mensajes mostrados en la tabla 3 permiten la lectura y la escritura de esos datos.

mensajes de lectura	mensajes de escritura
readBoolean()	writeBoolean(boolean)
readByte ()	writeByte(byte)
readShort()	writeShort(short)
readInt()	writeInt(int)
readLong()	writeLong(long)
readFloat()	writeFloat(float)
readDouble()	writeDouble(double)

Cuadro 3: Métodos para manejo de streams orientados a bytes con buffers para datos

La siguiente aplicación almacena en un stream orientado a bytes con buffers para datos un entero que corresponde al tamaño de un arreglo de dobles, las componentes de un arreglo de dobles y finalmente un valor de tipo `boolean`:

```
/**
 * Write data class Application
 */
```



```
import java.io.*;

class WriteData {

  public static void main(String[] args) {
    double data[] = { 10.3,20.65,8.45,-4.12 };

    FileOutputStream f;
    BufferedOutputStream bf;
    DataOutputStream ds;

    try {
      f = new FileOutputStream("file1.data");
      bf = new BufferedOutputStream(f);
      ds = new DataOutputStream(bf);

      ds.writeInt(data.length);
      for(int i = 0;i < data.length;i++)
        ds.writeDouble(data[i]);

      ds.writeBoolean(true);

      ds.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

Notar que el stream orientado a bytes con buffer para datos se crea en tres pasos: primero se crea un stream de salida, luego se asocia el buffer y finalmente se crea el stream de datos.

La siguiente aplicación lee los datos almacenados anteriormente utilizando un stream orientado a byte con buffer:

```
/**
 * Read data class Application
 */
import java.io.*;

class ReadData {

  public static void main(String[] args) {

    FileInputStream f;
```

```
    BufferedInputStream bf;
    DataInputStream ds;

    try {
      f = new FileInputStream("file1.data");
      bf = new BufferedInputStream(f);
      ds = new DataInputStream(bf);

      int length = ds.readInt();
      for(int i = 0;i < length;i++)
        System.out.println(ds.readDouble());

      System.out.println(ds.readBoolean());

      ds.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

La salida producida por la ejecución de la aplicación es:

```
10.3
20.65
8.45
-4.12
true
```

23.4. Streams orientados a caracteres

Los streams orientados a caracteres se utilizan para leer y escribir caracteres. Para crear streams de salida orientados a caracteres es necesario crear una instancia de un `FileWriter` y luego una instancia de un `BufferedWriter`. Existen tres métodos para escribir datos en esta clase de streams, los que son mostrados en la tabla 4.

mensaje
write(String,int,int)
write(char[],int,int)
newLine()

Cuadro 4: Métodos de escritura para streams orientados a caracteres

El primer mensaje se utiliza para escribir caracteres a partir de un string, iniciando la escritura a partir del caracter que ocupa la posición indicada por el

primer entero, y tantos caracteres como indique el segundo entero. El segundo mensaje es similar, pero los caracteres son tomados de un arreglo de caracteres. El mensaje `newLine` genera un caracter de fin de línea en el stream de salida, independientemente de la convención utilizada en el sistema operativo particular.

La siguiente aplicación escribe caracteres seleccionados de dos strings en un stream de salida orientado a caracteres:

```
/**
 * Write text class Application
 */
import java.io.*;

class WriteText {
  public static void main(String[] args) {
    FileWriter f;
    BufferedWriter bf;

    try {
      f = new FileWriter("file1.text");
      bf = new BufferedWriter(f);

      String s = "Hello World!";
      bf.write(s,0,s.length());
      bf.newLine();
      bf.write("Java is nice!!!",8,5);
      bf.newLine();

      bf.close(); }
     catch (IOException e) {
     System.out.println("Error with files:"+e.toString());
    }
  }
}
```

El contenido del archivo después de la ejecución de la aplicación es:

```
Hello World!
nice!
```

Para leer de un stream orientado a caracteres es necesario crear una instancia de un `FileReader` y luego una instancia de un `BuferedReader`. El mensaje `readLine` puede ser utilizado para leer líneas completas de un archivo de texto, retornando una instancia de un string que contiene la línea, o una referencia `null` si se ha alcanzado el final del archivo.

La siguiente aplicación lee líneas de un stream orientado a caracteres utilizando buffer:

```
/**
 * Read text class Application
 */
import java.io.*;

class ReadText {

  public static void main(String[] args) {

    FileReader f;
    BufferedReader bf;

    try {
      f = new FileReader("file1.text");
      bf = new BufferedReader(f);

      String s;
      while ((s = bf.readLine()) != null)
        System.out.println(s);

      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

23.5. Entrada estándar

En algunos casos las aplicaciones necesitan leer de la entrada estándar. En Java, la entrada estándar se referencia a través del objeto `System.in`. A fin de leer de ese objeto, es necesario crear una instancia de un `InputStreamReader` y de un `BufferedReader`, tal como muestra el siguiente ejemplo, el cual copia la entrada estándar en la salida estándar.

```
/**
 * Standard input class Application
 */
import java.io.*;

class StandardInput {

  public static void main(String[] args) {
    InputStreamReader isr;
    BufferedReader br;
```



```
    try {
      isr = new InputStreamReader(System.in);
      br = new BufferedReader(isr);

      String line;
      while ((line = br.readLine()).length() != 0)
        System.out.println(line);
    } catch(IOException e) {
      System.out.println("Error in standard input");
    }
  }
}
```

El método `readLine` retorna una línea de la entrada estándar como un string. Utilizando el método `length` se puede determinar si la entrada estándar ha sido cerrada.

La siguiente aplicación realiza la misma tarea que la aplicación anterior, pero ahora el tratamiento de la excepción `IOException` es diferente:

```
/**
 * Standard input class Application (throws IOException)
 */
import java.io.*;

class StandardInputWithThrows {

  public static void main(String[] args) throws IOException {
    InputStreamReader isr;
    BufferedReader br;

    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);

    String line;
    while ((line = br.readLine()).length() != 0)
      System.out.println(line);
  }
}
```

Notar que en el ejemplo no se ha utilizado el bloque `try catch` a pesar que es obligatorio tratar la excepción de entrada salida. Esto puede ser hecho si el método (`main` en este caso) utiliza la palabra clave `throws` en su definición, manifestando su interés en no tratar la excepción. Si se produce una excepción de entrada salida, el método desde donde se invocó esta operación deberá estar en condiciones de tratar con ella. En este caso particular, dado que el método es la función `main`, el shell recibirá la excepción y la ejecución de la aplicación finalizará con un mensaje de error.

El manejo de la entrada estándar es complejo en Java, por eso a partir de la versión 5.0 (JDK 1.5.0), se simplifica el manejo de la entrada estándar a través de la clase `Scanner`. Utilizando esta clase, se puede crear un objeto que lea de la entrada estándar en una forma mucho más simple. El ejemplo siguiente muestra una aplicación que lee números enteros de la entrada estándar e imprime su suma:

```
/**
 * Sum integers read from System.in with the Scanner class
 */
import java.io.*;
import java.util.*;

class ReadWithScanner {

  public static void main(String[] args) throws IOException {

      Scanner sc = new Scanner(System.in);
      int sum = 0;
      while (sc.hasNextInt()) {
          int anInt = sc.nextInt();
           sum += anInt;
      }
      System.out.println(sum);
  }
}
```

El objeto scanner lee y particiona la entrada en tokens usando delimitadores, los que por defecto son separadores (espacios en blanco, caracteres columnas, newline, etc.). Los tokens resultantes son convertidos a valores de distintos tipos usando los métodos `next`.

Los objetos de la clase `Scanner` pueden ser utilizados para leer datos desde archivos tal como se muestra a continuación:

```
Scanner from_file = new Scanner(new File("file.data"));
```

El lector interesado puede recurrir a la documentación de Java 5.0 API para más detalles.

24. Threads

En Java se pueden definir tareas que se ejecutan en forma concurrente. Cada una de estas tareas, que se denominan *threads*, es una tarea que se ejecuta en forma independiente, con un determinado tiempo de CPU asignado a ella. Las threads se pueden comunicar entre ellas y acceder a datos compartidos, acceso que deberá ser sincronizado.

Para definir una thread, es necesario crear una subclase de la clase `Thread`. Esta clase tiene un método abstracto llamado `run` que debe ser definido en la subclase. Este método contiene el código que será ejecutado como tarea independiente.

El siguiente ejemplo define una clase denominada `CharThread` que es subclase de `Thread`, y por lo tanto debe definir el método `run`:

```
/**
 * Char thread class Application
 */


class CharThread extends Thread {
  char c;

  CharThread(char aChar) {
    c = aChar;
  }

  public void run() {
    while (true) {
      System.out.println(c);
      try {
        sleep(100);
      } catch (InterruptedException e) {
        System.out.println("Interrupted");
      }
    }
  }
}
```

Esta clase define un campo de tipo caracter que se inicializa con un determinado valor cuando se crea una instancia de la clase con el constructor. El método `run` tiene un ciclo infinito, donde el caracter es impreso en la salida estándar, y la thread es suspendida por 100 milisegundos. Notar que se puede generar una excepción cuando la thread está suspendida que debe ser tratada.

La siguiente aplicación crea dos instancias de la clase, cada una inicializada con un caracter distinto. Luego ambas threads son puestas en ejecución, lo que significa que los métodos `run` de ambas instancias son ejecutados concurrentemente:

```
/**
 * test threads class Application
 */

class TestThreads {
```

```
  public static void main(String[] args) {

    CharThread t1 = new CharThread('a');
    CharThread t2 = new CharThread('b');

    t1.start();
    t2.start();
  }
}
```

Notar que las dos instancias de la clase `CharThread` se crean llamando al constructor. Ambas threads son iniciadas en su ejecución cuando se envía el mensaje `start`. La salida producida por la ejecución de la aplicación será similar a:

```
a
b
a
b
a
b
a
b
a
b
a
b
...
```

Ambas threads obtienen la CPU por algún instante de tiempo, por lo que la salida que se produce es la combinación de la salida de ambas threads.

24.1. Productor y consumidor

El problema del productor y consumidor es un ejemplo estándar que ilustra la concurrencia y los problemas asociados con ella. La idea es que existen dos procesos que interactúan a través de un buffer común en el que el productor almacena elementos producidos y el consumidor los toma consumiéndolos. En este caso surge un problema de sincronización dado que ambos procesos deben interactuar a través del mismo buffer. El buffer constituye entonces lo que se denomina un *recurso crítico*. El otro problema que surge es que el productor no puede colocar elementos en un buffer lleno y el consumidor no puede consumir elementos de un buffer vacío.

La siguiente aplicación muestra la posible implementación de la clase principal del ejemplo del productor y consumidor:


```
/**
 * Producer Consumer class Application
 */

class ProducerConsumer {

  public static void main(String[] args) {

    Buffer buffer = new Buffer(20);

    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);

    prod.start();
    cons.start();
  }
}
```

Esta clase crea un buffer con lugar para contener 20 elementos, luego una instancia de la thread que corresponde al productor, y luego una instancia de la thread que corresponde al consumidor. El buffer común es pasado como argumento en los constructores, de forma tal que ambas threads puedan compartir el buffer. Finalmente, ambas threads son puestas en ejecución.

La clase correspondiente al productor se puede definir como se muestra a continuación:

```
/**
 * Producer class Application
 */
class Producer extends Thread {
  Buffer buffer;

  public Producer(Buffer b) {
    buffer = b;
  }

  public void run() {
    double value = 0.0;

    while (true) {
      buffer.insert(value);
      value += 0.1;
    }
  }
}
```

La clase productor es una subclase de `Thread`, por lo que debe definir el método `run` que contiene el código que se ejecuta como tarea independiente. La clase define un campo que contendrá una referencia al buffer pasado como argumento en el constructor. El método `run` inserta un valor doble en el buffer dentro de un ciclo infinito. De esta forma el productor produce continuamente valores dobles que son insertados en el buffer común

Similarmente, el consumidor puede ser definido como sigue:

```
/**
 * Consumer class Application
 */

class Consumer extends Thread {
  Buffer buffer;

  public Consumer(Buffer b) {
    buffer = b;
  }

  public void run() {

    while(true) {
      System.out.println(buffer.delete());
    }
  }
}
```

La clase consumidor también es subclase de `Thread`. La clase define también un campo para contener una referencia al buffer común. El método `run` simplemente en forma repetida elimina los elementos del buffer y los imprime en la salida estándar.

El buffer en este ejemplo se define como un buffer circular implementado con un arreglo y dos enteros que actúan como punteros, uno para indicar el frente de la cola, y el otro el final. Los datos se insertan por la cola y se leen por la cabeza (frente). También se define un campo usado para almacenar la cantidad de elementos disponibles en el buffer. La clase que define el buffer se muestra a continuación:

```
/**
 * Buffer class Application
 */

class Buffer {

  double buffer[];
```



```
  int head = 0;
  int tail = 0;
  int size = 0;
  int numElements = 0;

  public Buffer(int s) {
    buffer = new double[s];
    size = s;
    numElements = 0;
  }

  public void insert(double element) {

    buffer[tail] = element;
    tail = (tail + 1) % size;
    numElements++;
  }

  public double delete() {

    double value = buffer[head];
    head = (head + 1) % size;
    numElements--;
    return value;
  }
}
```

Aunque la implementación a primera vista parece correcta, no funciona por las siguientes dos razones:

- Ambos métodos `insert` y `delete` operan concurrentemente sobre la misma estructura. Es necesario definir entonces una región crítica, o dicho en otros términos, prevenir que ambos métodos puedan acceder al buffer en forma concurrente: si una thread está insertando datos, la otra thread debe esperar hasta que la primera termine y viceversa.
- El método `insert` no verifica que haya al menos un lugar libre en el buffer antes de insertar, y el método `delete` no verifica que haya al menos un elemento en el buffer para eliminar.

Las siguientes dos secciones analizan estos problemas y plantean las soluciones respectivas.

24.2. Métodos sincronizados

En Java se pueden definir métodos sincronizados. Estos métodos no pueden ejecutarse en forma concurrente sobre una misma instancia. Cada instancia tiene un elemento denominado *lock* que se utiliza para la sincronización. Cuando

una thread comienza la ejecución de un método sincronizado toma el lock, no permitiendo que otras threads ejecuten métodos sincronizados. Cuando el método sincronizado termina su ejecución, libera el lock.

La solución al primer problema consiste en definir los métodos como se muestra a continuación:

```
public synchronized void insert(double element) {

  buffer[tail] = element;
  tail = (tail + 1) % size;
  numElements++;
}

public synchronized double delete() {

  double value = buffer[head];
  head = (head + 1) % size;
  numElements--;
  return value;
}
```

24.3. Wait y notify

Las subclases de `Thread` pueden enviar los mensajes `wait` y `notify`. Estos mensajes se pueden enviar solamente desde métodos sincronizados. El mensaje `wait` suspende la thread que lo envía, liberando el lock. El mensaje `notify` despierta una thread que está esperando la liberación del lock.

En el ejemplo del buffer, una thread que está por insertar un elemento en el buffer debe suspenderse a si misma si encuentra que éste está lleno. De la misma forma, una thread que está por remover un elemento del buffer debe suspenderse a si misma si el buffer está vacío.

La thread que inserta elementos en un buffer vacío debe por otra parte, notificara la thread que los elimina que hay elementos para remover, y la thread que elimina elementos de un buffer lleno, debe notificar a la thread que inserta que ahora hay lugar para insertar más elementos.

El código correcto para la implementación de los métodos `insert` y `delete` se muestra a continuación:

```
public synchronized void insert(double element) {
  if (numElements == size) {
    try {
      wait();}
     catch(InterruptedException e) {
     System.out.println("Interrupted");
    }
  }
```



```
  buffer[tail] = element;
  tail = (tail + 1) % size;
  numElements++;
  notify();
}

public synchronized double delete() {

  if (numElements == 0) {
    try {
      wait();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
  double value = buffer[head];
  head = (head + 1) % size;
  numElements--;
  notify();
  return value;
}
```

Notar que es necesario tratar la excepción correspondiente cuando se utiliza el método `wait` para considerar el caso en que la tarea sea despertada por otra razón. El listado completo de la clase `Buffer` se presenta en el Apéndice D.

25. Archivos JAR

Cuando se compila el ejemplo del productor y consumidor, se generan cuatro archivos que contienen cada una de las clases:

```
# ls *.class
Buffer.class
Consumer.class
ProducerConsumer.class
Producer.class
```

Si se desea distribuir esta aplicación es necesario copiar los cuatro archivos, indicando además cual es la clase principal (aquella que contiene el método `main`). Sin embargo, Java provee un mecanismo para empaquetar y comprimir los archivos en un único archivo con el objeto de hacer la distribución de aplicaciones más simple. Este archivo comprimido recibe el nombre de JAR file (Java ARchive).

Un archivo JAR se crea y manipula con el comando `jar`. Para crear el archivo JAR, es necesario definir un archivo *manifest*. Este archivo contiene información

sobre los archivos que deben ser incluidos en el archivo JAR. El comando `jar` crea un archivo manifest por defecto en el directorio `META-INF` con el nombre `MANIFEST.MF`, justo bajo el directorio corriente.

Se pueden agregar líneas específicas a este archivo pasando como un argumento del comando `jar` el nombre de un archivo de texto que contiene estas líneas. La información se especifica en pares (*clave*,*valor*). En el ejemplo del productor y consumidor, el único par necesario para incorporar es el que especifica el nombre de la clase que contiene la función `main`. Esto puede ser hecho en un archivo (llamado *mylines.txt* en este ejemplo) con el siguiente contenido:

```
# cat mylines.txt
Main-Class: ProducerConsumer
```

La creación del archivo JAR para esta aplicación se hace entonces:

```
# jar cmf mylines.txt ProducerConsumer.jar
  ProducerConsumer.class Producer.class Consumer.class
  Buffer.class
```

La opción `c` especifica que se debe crear un archivo JAR, la opción `m` que el nombre del archivo de texto que contiene las líneas a incorporar al manifest file será pasado en la línea de comandos, y la opción `f` indica que el nombre del archivo JAR que se desea obtener también será pasado en la línea de comandos. El archivo `mylines.txt` contiene las líneas para el archivo manifest, `ProducerConsumer.jar` es el nombre esperado del archivo de salida, y los nombres que siguen son los nombres de los archivos que deben ser agregados al archivo JAR.

Es posible listar el contenido del archivo JAR usando la opción `t` como se muestra a continuación:

```
# jar tf ProducerConsumer.jar
META-INF/
META-INF/MANIFEST.MF
ProducerConsumer.class
Producer.class
Consumer.class
Buffer.class
```

Notar que el archivo manifest contiene las líneas que fueron especificadas:

```
Manifest-Version: 1.0
Main-Class: ProducerConsumer
Created-By: 1.4.0 (Sun Microsystems Inc.)
```

La aplicación incluida en el archivo JAR se puede ejecutar entonces como se muestra a continuación:

```
# java -jar ProducerConsumer.jar
```

También es posible extraer y actualizar los contenidos de un archivo JAR, consulte la documentación para conocer más detalles.

A. La clase Book

```
/**
 * Books Application
 */

class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;
  static String owner;

  /** This constructor creates a Book with a specified title,
   *  author, number of pages and unknown ISBN
   */

  Book(String tit,String aut,int num) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = "unknown";
  }

  /** This constructor creates a Book with a specified title,
   *  author, number of pages and ISBN
   */

  Book(String tit,String aut,int num,String isbn) {
    title = tit;
    author = aut;
    numberOfPages = num;
    ISBN = isbn;
  }

  /** This method returns a string containing the initials of
   *  the author
   */

  public String getInitials() {
    String initials = "";

    for(int i = 0;i < author.length();i++) {
      char currentChar = author.charAt(i);
      if (currentChar >= 'A' && currentChar <='Z') {
        initials = initials + currentChar + '.';
```

```
      }
    }
    return initials;
  }

  /** This method returns true if both the receptor and the
   *  argument correspond to the same book
   */

  public boolean equals(Book b) {
    return (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }

  /** This method sets the owner of the book
   */

  public void setOwner(String name) {
    owner = name;
  }

  /** This method gets the owner of the book
   */

  public String getOwner() {
    return owner;
  }

  /** This method returns a description of the book
   */

  public static String description() {
    return "Book instances can store information on books";
  }
}
```


B. La clase Complex

```
/**
 * Complex Number class
 */

public class Complex {
  private double real;         // real part
  private double im;           // imaginary part

  /** This constructor creates a complex number from its real
   *  and imaginary part.
   */

  Complex(double r,double i) {
    real = r;
    im = i;
  }

  /** This constructor creates a complex number as a copy
   *  of the complex number passed as argument
   */

  Complex(Complex c) {
      this(c.real,c.im);
  }

  /** This method returns the real part
   */

  public double getReal() {
    return real;
  }

  /** This method returns the imaginary part
   */

  public double getImaginary() {
    return im;
  }

  /** This method returns a new complex number wich is
   *  the result of the addition of the receptor and the
   *  complex number passed as argument
   */
```

```
  public Complex add(Complex c) {
    return new Complex(real + c.real,im + c.im);
  }

  /** This method returns a new complex number wich is
   *  the result of the substraction of the receptor and the
   *  complex number passed as argument
   */

  public Complex sub(Complex c) {
    return new Complex(real - c.real,im - c.im);
  }

  /** This method returns a new complex number wich is
   *  the result of the product of the receptor and the
   *  complex number passed as argument
   */

  public Complex mul(Complex c) {
    return new Complex(real * c.real - im * c.im,
                       real * c.im + im * c.real);
  }

  /** This method returns a new complex number wich is
   *  the result of the product of the receptor and the
   *  complex number passed as argument
   */

  public Complex div(Complex c) {
    double r,i;

    if (Math.abs(c.real) >= Math.abs(c.im)) {
      double n = 1.0 / (c.real + c.im * (c.im / c.real));
      r = n * (real + im * (c.im / c.real));
      i = n * (im - real * (c.im / c.real));
    } else {
      double n = 1.0 / (c.im + c.real * (c.real / c.im));
      r = n * (im + real * (c.real / c.im));
      i = n * (- real + im * (c.real / c.im));
    }
    return new Complex(r,i);
  }

  /** This method returns a new complex number wich is
   *  the result of the scaling the receptor by the
   *  argument
```



```
   */

  public Complex scale(double c) {
    return new Complex(real * c,im * c);
  }

  /** This method computes the norm of the receptor
   */

  public double norm() {
    return Math.sqrt(real * real + im * im);
  }

  /** This method increments the real part by a value
   *  passed as argument. Note that the method modifies
   *  the receptor
   */

  public Complex addReal(double c) {
    real += c;
    return this;
  }

  /** This method returns a string representation of
   *  the receptor
   */

  public String asString() {
    return "" + real + " + i * " + im;
  }
}
```

C. La clase ScientificBook

```
/**
 * Scientific Book Class
 */

class ScientificBook extends Book {
  String area;
  boolean proceeding = false;

  /** This constructor creates a Scientific Book with a
   *  specified title, author, number of pages, ISBN and
   *  area. Proceeding is set to false
   */

  ScientificBook(String tit,String aut,int num,String isbn,
                 String a) {
    super(tit,aut,num,isbn);
    area = a;
  }

  /** This method returns true if both the receptor and the
   *  argument correspond to the same book
   */

  public boolean equals(ScientificBook b) {
    return super.equals(b) && area.equals(b.area) &&
           proceeding == b.proceeding;
  }

  /** This method returns a description of the book
   */

  public static String description() {
    return "ScientificBook instances can store information" +
           " on scientific books";
  }

  /** This method sets proceeding to true
   */

  public void setProceeding() {
    proceeding = true;
  }

  /** This method sets proceeding to false
```



```
   */

  public boolean isProceeding() {
    return proceeding;
  }
}
```

D. El ejemplo del productor y consumidor

```
/**
 * Producer Consumer class Application
 */

class ProducerConsumer {

  /** This method creates a common buffer, and starts two
   *  threads: the producer and the consumer
   */

  public static void main(String[] args) {

    // creates the buffer
    Buffer buffer = new Buffer(20);

    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);

    // start the threads
    prod.start();
    cons.start();
  }
}

/**
 * Producer class Application
 */

class Producer extends Thread {
  Buffer buffer;

  /** This constructor initialize the data member buffer as
   *  a reference to the common buffer
   */

  public Producer(Buffer b) {
    buffer = b;
  }

  /** This method executes as a thread. It keeps inserting
   *  a value into the buffer
   */

  public void run() {
```



```
    double value = 0.0;

    while (true) {
      buffer.insert(value);
      value += 0.1;
    }
  }

/**
 * Consumer class Application
 */

class Consumer extends Thread {
  Buffer buffer;

  /** This constructor initialize the data member buffer as
   *  a reference to the common buffer
   */

  public Consumer(Buffer b) {
    buffer = b;
  }

  /** This method executes as a thread. It keeps removing
   *  values from the buffer, and printing them
   */

  public void run() {

    while(true) {
    System.out.println(buffer.delete());
    }
  }
}

/**
 * Buffer class Application
 */

class Buffer {

  double buffer[];

  int head = 0;
  int tail = 0;
  int size = 0;
```

```
  int numElements = 0;

  /** This constructor initialize the data member buffer as
   *  an array of doubles. The size of the array is also
   *  initialized
   */

  public Buffer(int s) {
    buffer = new double[s];
    size = s;
    numElements = 0;
  }

  /** This method inserts an element into the circular
   *  buffer. The thread goes to sleep if there is no empty
   *  slots, and notify waiting threads after inserting an
   *  element
   */

  public synchronized void insert(double element) {

    if (numElements == size) {
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("Interrupted");
      }
    }
    buffer[tail] = element;
    tail = (tail + 1) % size;
    numElements++;
    notify();
  }

  /** This method removes an element from the circular
   *  buffer. The thread goes to sleep if there is no element
   *  to remove, and notify waiting threads after removing an
   *  element
   */

  public synchronized double delete() {

    if (numElements == 0) {
      try {
        wait();
      } catch(InterruptedException e) {
```



```
        System.out.println("Interrupted");
      }
    }
    double value = buffer[head];
    head = (head + 1) % size;
    numElements--;
    notify();
    return value;
  }
}
```

Referencias

[1] Eckel B, ***Thinking in Java (3rd Edition)*** Prentice Hall. 2002. Available on-line at `http://jamesthornton.com/eckel/`

[2] Flanagan D, ***Java in a Nutshell: A desktop quick reference (3rd Edition)***, O'Reilly, 1999.

[3] Holzner S, ***Java 2***, The Coriolis Group, 2000.

[4] Hortsmann C, ***Computing concepts with Java 2, essentials (2nd Edition)***, Wiley, 2000.

[5] Horstmann C and Cornell G, ***Core Java, Volume II – Advanced Features***, The Sun Microsystems Press, 2000.

[6] Horstmann C and Cornell G, ***Core Java, Volume I – Fundamentals***, The Sun Microsystems Press, 2001.

[7] Lemay L and Cadenhead R, ***Java 2. Guida Completa***. Apogeo s.r.l., 2000.

[8] Sun Microsystems, ***The Java tutorial. A practical guide for programmers***. Available online at `http://java.sun.com`