Universidad Autónoma de Baja California
Facultad de Ciencias Químicas e Ingeniería
Ingeniería en Computación

Apuntes del curso **Programación orientada a objetos I**

Basados en los libros “Object-Oriented Technology” de David A. Taylor, Addison-Wesley, 1990 y “Programming with Java” de John R. Hubbard, Serie Schaum’s, McGraw Hill, 1999.

Dr. Guillermo Licea Sandoval
Profesor titular de tiempo completo

Contenido del curso

Bibliografía

Object-oriented technology. David A. Taylor. Addison-Wesley, 1990.
Programming with Java. John R. Hubbard. Mc Graw-Hill, 1999.
Data Structures with Java. John R. Hubbard. McGraw-Hill, 2001.
Java in a nutshell, second edition. O'Reilly, 1999.
Java 2 complete. Sybex, 1999.

Software

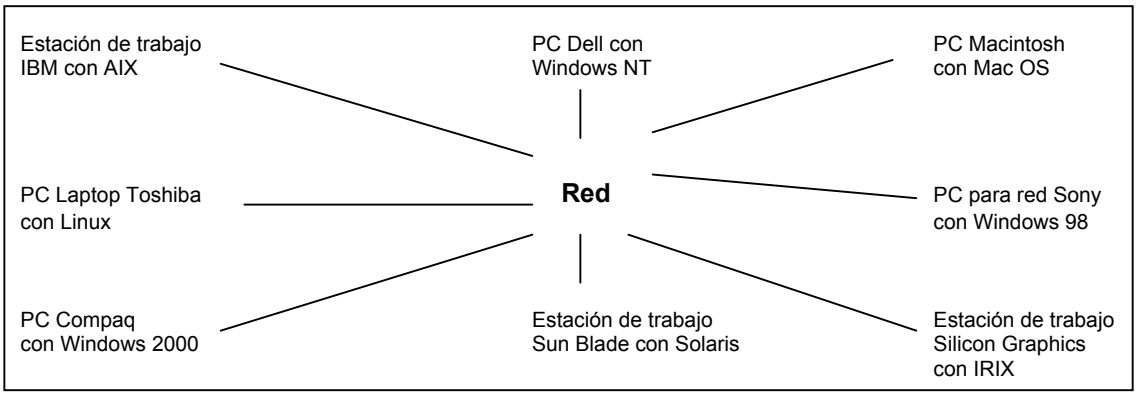
Java Development Kit (JDK), versión 1.2 o mayor y editor de textos (Notepad, Win edit, etc.)

1. Aspectos básicos de Java

1.1.¿ Qué es el lenguaje Java ?

El lenguaje Java fue desarrollado por James Gosling en la compañía Sun Microsystems a partir de 1991. El nombre del lenguaje proviene de un populismo para nombre el café. Cuando apareció la World Wide Web tuvo su auge en 1993, el lenguaje fue mejorado para facilitar la programación en la Web y fue lanzado al mercado a finales de 1995. Desde entonces se ha convertido en uno de los lenguajes más populares, especialmente para la programación de aplicaciones de redes.

Para entender porque Java es el lenguaje que eligen los programadores de aplicaciones para redes, imaginemos una red de computadoras heterogéneas.

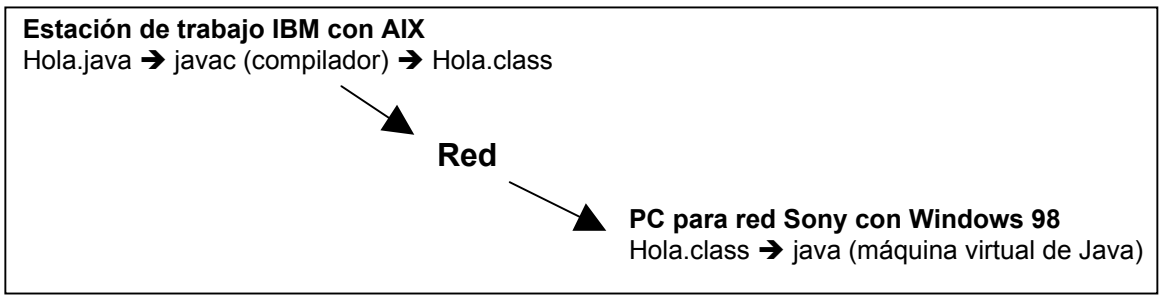


Las computadoras que se muestran en la figura anterior podrían estar en una red local, o tal vez en diferentes ciudades, países o continentes. Lo principal es que funcionan con diferentes sistemas operativos y posiblemente con diferentes microprocesadores.

Si escribiéramos un programa para la estación de trabajo IBM, este programa no funcionaría en las otras computadoras debido a que el programa es traducido por un compilador al lenguaje máquina de la computadora, este lenguaje esta relacionado con el microprocesador y por tanto varía de una computadora a otra.

Para resolver este problema, el lenguaje Java provee un compilador y un sistema llamado la máquina virtual de Java (JVM) para cada tipo de computadora. El compilador de Java traduce el código Java a un lenguaje intermedio llamado código de bytes (bytecode). El código de bytes es independiente del tipo de computadora, el mismo código de bytes puede ser utilizado por cualquier computadora que pueda ejecutar la máquina virtual de Java.

La figura siguiente muestra la misma red de computadoras heterogéneas. Aquí se muestra como un programa escrito para la estación de trabajo IBM puede ser ejecutado en una PC para red Sony aun cuando funcionan con diferente sistema operativo.



1.2.¿ Dónde conseguir y como instalar el JDK ?

El JDK (Java Development Kit) es una colección de programas para ayudar a los programadores a compilar, ejecutar y depurar programas escritos en Java. No es tan bueno como un ambiente integrado de desarrollo (IDE), pero contiene lo suficiente para iniciar con la programación en Java. El JDK es distribuido gratuitamente por Sun Microsystems. La dirección del sitio es http://www.sun.com. Sólo se necesita presionar en la liga Products & APIs y después presionar en la versión del JDK que se quiere obtener. Una vez que se tiene el archivo con el JDK se requiere ejecutar el archivo para realizar la instalación.

1.3.¿ Cómo crear y ejecutar el programa hola mundo con el JDK bajo Windows ?

Para crear un programa en Java se requiere un editor de textos:

1) Se escribe el programa
2) Se escribe un archivo por cada clase
3) El nombre de la clase y del archivo deben ser iguales
4) La extensión del archivo debe ser `.java`

Ejemplo 1.1: el programa `Hola.java`

```
public class Hola {
      public static void main(String[] args) {
            System.out.println("Hola mundo");
      }
}
```

La primera línea declara una clase llamada Hola. Cualquier programa en Java inicia así, el nombre de la clase puede ser cualquiera que cumpla con la sintaxis. El nombre de la clase debe ser igual al del archivo y debe tener la extensión *`.java`*. Al final de la primera línea se encuentra una llave *`{`*, está debe aparecer inmediatamente después del nombre de la clase.

La segunda línea contiene las palabras `public`, `static`, `void` y `main`. La palabra `public` significa que el contenido del siguiente bloque está accesible para las otras clases. La palabra `static` significa que el método que se está definiendo pertenece a la clase misma y no a los objetos de la clase. La palabra `void` indica que el método que se define no regresará ningún valor. La palabra `main` indica el nombre del método.

La cadena que aparece entre paréntesis al final de la segunda línea es la lista de parámetros del método `main()`. Esta lista declara los parámetros que son variables locales usadas para transmitir datos al método desde el exterior. La lista de parámetros para el método `main()` siempre es igual.

La tercera línea contiene un enunciado que indica al sistema que despliegue en la pantalla el mensaje Hola mundo. Este mensaje es una cadena de caracteres y por eso debe ser escrito entre comillas. La palabra `println()` es el nombre del método que indica al sistema como desplegar.

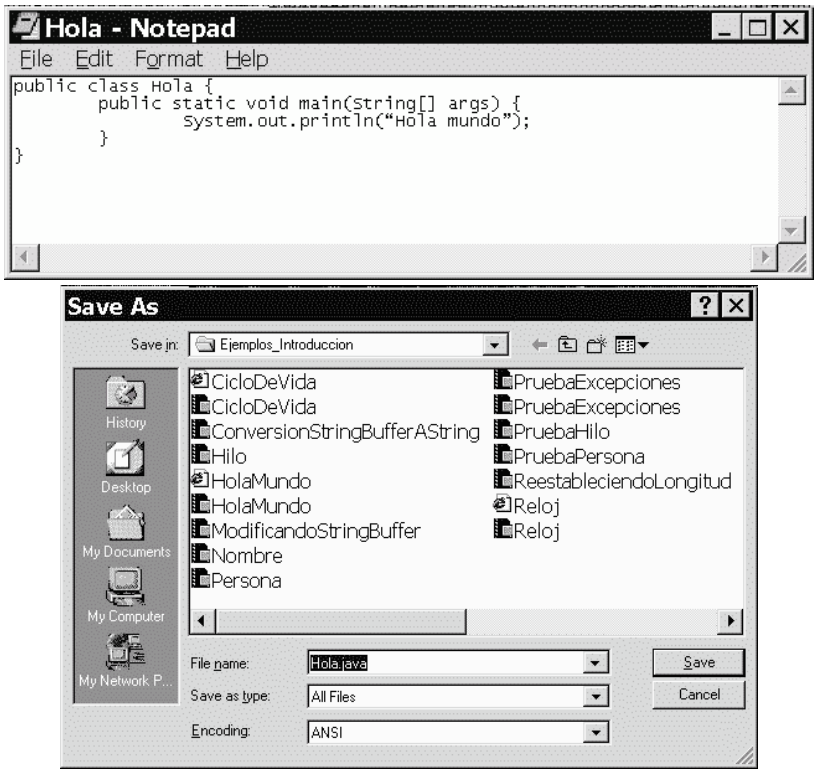
`System.out` es el nombre de un objeto que pertenece a la clase donde fue definido el método `println()`. Este objeto es quien recibe la petición de despliegue.

Para crear que contendrá el programa Java se puede utilizar la aplicación `Notepad` de `Windows`, cuidando de darle al archivo el mismo nombre que tenga la clase principal.

Una vez creado el o lo archivos que contienen las clases que componen el programa, se compila utilizando el compilador del JDK.

Para llevar a compilación se abre una ventana de comandos de DOS y se teclea:

```
javac Hola.java
```



Si existen errores en el programa aparecerán inmediatamente y se requerirá corregirlos para volver a compilar. Cuando ya no existan errores se podrá ejecutar el programa a través del interprete (máquina virtual de Java):

```
java Hola
```

1.4. Los comentarios en Java

Hay dos maneras de escribir comentarios en Java. Un comentario al estilo de C empieza con los símbolos `/*` y termina con `*/`. Un comentario estilo C++ empieza con los símbolos `//` y termina al finalizar la línea.

Los comentarios estilo C pueden utilizarse en la misma línea como:

```
public /* acceso / class /* declaración */ Hola
```

Lo convencional es utilizar los comentarios estilo C para comentarios que requieren varias líneas. Los comentarios estilo C++ se utilizan para hacer anotaciones al final de una línea de código.

Por ejemplo en el programa anterior podríamos agregar comentarios.

```
/* Este programa despliega una linea con el mensaje
   Hola mundo */

public class Hola {   // Declaracion de la clase
      public static void main(String[] args) { // Metodo principal
            System.out.println("Hola mundo");   // Mensaje
      }
}
```

1.5. La entrada de datos

La entrada de datos es más sensible a los errores, si la entrada de datos no es del tipo correcto, el programa falla. Este tipo de error se llama *excepción*. Java provee mecanismos especiales para manejar las excepciones. La forma más simple de manejar las excepciones de entrada/salida es a través de la cláusula *throws IOException* sólo hay que agregar está cláusula a la declaración del método *main*.

Ejemplo 1.2: Entrada interactiva

```
import java.io.*;

public class EntradaInteractiva {
      public static void main(String[] args) throws IOException {
            InputStreamReader lector = new InputStreamReader(System.in);
            BufferedReader entrada = new BufferedReader(lector);
            System.out.println("Escribe tu nombre: ");
            String nombre = entrada.readLine();
            System.out.println("Hola, "+ nombre + ".");
      }
}
```

Este programa despliega en la pantalla el mensaje “Escribe tu nombre: “ y espera una entrada de datos. Cuando el usuario escribe su nombre y presiona la tecla Enter, el sistema despliega “Hola, y el nombre del usuario”.

La primera línea del programa es `import java.io.*;` Esto le dice al compilador que busque en el paquete (biblioteca) `java.io` la definición de las clases de entrada/salida que se utilizan en el programa.

La cuarta línea define el objeto *lector* como una instancia de la clase `InputStreamReader`, atándola al flujo de entrada del sistema `System.in`. Esto significa que el objeto *lector* servirá como un conducto del teclado al programa.

La quinta línea define al objeto `entrada` como una instancia de la clase `BufferedReader`, atándola al objeto `lector`. Esto significa que el objeto entrada puede ser usado para extraer el dato de entrada de una manera fácil. En particular, es posible utilizar el método `readLine()` para leer una cadena de caracteres completa desde el teclado y guardarla en un objeto `String`, esto se hace en la séptima línea del programa: `String`

`nombre = entrada.readLine();`. Esto declara el objeto `nombre` de tipo `String` y es inicializado al valor regresado por el método `entrada.readLine()`. El resultado es que el objeto `nombre` contendrá lo que el usuario teclee.

La expresión `"Hola, " + nombre + "."` Significa que se deben concatenar las tres cadenas para formar una nueva.

1.6. La entrada numérica

La entrada en el ejemplo anterior fue un `String`: una cadena de caracteres. Las cadenas de caracteres y los números son procesados de manera distinta. Por ejemplo el operador `+` funciona sobre ambos tipos, pero con resultados muy distintos. En el caso de números se realiza una suma y en el caso de cadenas de caracteres se realiza una concatenación.

Ejemplo 1.3: Cálculo del año de nacimiento

```
import java.io.*;

public class Nacimiento {
      public static void main(String[] args) throws IOException {
            InputStreamReader lector = new InputStreamReader(System.in);
            BufferedReader entrada = new BufferedReader(lector);
            System.out.println("Escribe tu edad: ");
            String cadena = entrada.readLine();
            int edad = new Integer(cadena).intValue();
            System.out.println("Tu tienes " + edad + "años ahora");
            int ano = 2002 – edad;
            System.out.println("Probablemente naciste en " + ano);
      }
}
```

Al igual que el programa anterior también se importan clases del paquete `java.io`. También se incluye la cláusula `throws IOException` en el método `main()` para usar el método `readLine()`. Después de leer la entrada como un `String`, se convierte el dato en un valor entero con la expresión `new Integer(text).intValue()`. Este valor entero se utiliza para iniciar la variable `edad` que es de tipo `int`. El programa despliega ese valor, calcula el año de nacimiento y lo despliega también.

Ejemplo 1.4: Cálculo del área de un círculo

```
import java.io.*;

public class Area {
      public static void main(String[] args) throws IOException {
            InputStreamReader lector = new InputStreamReader(System.in);
            BufferedReader entrada = new BufferedReader(lector);
            System.out.println("Escribe el radio: ");
            String cadena = entrada.readLine();
            Double x = new Double(cadena);
            double radio = x.doubleValue();
            System.out.print("El area del circulo de radio " + radio);
```

```
            double area = Math.PI * radio * radio;
            System.out.println(" es " + area);
        }
}
```

Este programa recibe el radio de un círculo y despliega su área. La estructura del programa es muy parecida a los dos ejemplos anteriores. La diferencia principal está en el uso del objeto `x` y las variables `radio` y `area`.

El objeto `x` es una instancia de la clase `Double`. La variable `radio` es del tipo `double`. Ambas representan el mismo número real. La razón por la cual tenemos dos objetos que representan la misma cosa es por las diferentes operaciones que podemos realizar sobre los valores. El objeto `x` por ser una instancia de la clase `Double` puede obtener su valor directamente del objeto `cadena`.

1.7. Las variables y los objetos

En Java existen dos tipos de entidades que almacenan datos: las variables y los objetos. Una variable tiene tipo y almacena un valor sencillo. Un objeto es una instancia de una clase y puede contener muchas variables cuyos valores determinan el estado del objeto. Existen nueve tipos de variables posibles, pero los programadores pueden definir sus propias clases y por tanto puede existir una cantidad ilimitada de objetos. Cada variable tiene un nombre único que es designado al momento de la declaración. Los objetos tienen referencias en vez de nombres y no tienen que ser únicas. Una variable es creada al momento de la declaración y permanece viva mientras exista el método en el cual fue creada. Un objeto se crea usando el operador `new` para invocar a un constructor y muere cuando ya no tiene referencia.

Una referencia es una variable cuyo tipo es referencia a alguna clase. En el ejemplo 1.4, cadena es una referencia a la clase `String`. Como cualquier variable cada referencia tiene un tipo.

Además de los tipos referencia, existen ocho tipos de datos en Java. Estos son llamados tipos primitivos para distinguirlos de los tipos referencia.

```
bolean      false o true
char        caracteres de código único de 16 bits
byte        enteros de 8 bits (de -128 a 127)
short       enteros de 16 bits (de -32,768 a 32,767)
int         enteros de 32 bits (de -2,147,483,648 a 2,147,483,647)
long        enteros de 64 bits (de -9,223,372,036,854,775,807 a
            9,223,372,036,854,775,807)
float       reales de 32 bits
double      reales de 64 bits
```

La sintaxis para declarar una variables es:

```
nombre-del-tipo nombre-de-la-variable;
```

Todas las variables deben ser declaradas antes de usarse y pueden ser iniciadas con algún valor al momento de la declaración:

nombre-del-tipo nombre-de-la-variable = valor-inicial;

Ejemplo 1.5: Tipos de datos primitivos

```
public class DespliegaTiposDeDatos {
      public static void main(String[] args) {
            boolean b = false;
            char c = 'R';
            byte j = 127;
            short k = 32767;
            int m = 2147483647;
            long n = 922337203685477580 7L;       // L es por long
            float x = 3.14159265F;         // F es por flota
            double y = 3.141592653589793238;

            System.out.println("b = " + b);
            System.out.println("c = " + c);
            System.out.println("j = " + j);
            System.out.println("k = " + k);
            System.out.println("m = " + m);
            System.out.println("n = " + n);
            System.out.println("x = " + x);
            System.out.println("y = " + y);
      }
}
```

Para los valores literales de tipo long y flota es necesario escribir las letras `L` y `F` al final de la literal, respectivamente.

1.8. La aritmética y los operadores

Un operador es una función que tiene un nombre simbólico especial y es invocado a través de ese símbolo en una expresión.

Los operadores aritméticos y de asignación de Java son los siguientes:

```
Aritméticos:                +,  -,  *,  /,  %
De asignación:              +=,  -=,  *=,  /=,  %=,  &=,  ^=,  |=
Incremento/decremento:      ++,  --
```

Ejemplo 1.6: Operadores aritméticos, de incremento y de decremento

```
public class Aritmetica {
      public static void main(String[] args) {
            char c = 'R';
            byte j = 127;
            short k = 32767;
            int m = 25, n = 7;

            System.out.println("c = " + c);
            ++c;
            System.out.println("c = " + c);
```

```
                System.out.println("j = " + j);
                --j;
                System.out.println("j = " + j);
                ++j;
                System.out.println("j = " + j);
                ++j;
                System.out.println("j = " + j);

                System.out.println("k = " + k);
                K -= 4;
                System.out.println("k = " + k);
                K += 5;
                System.out.println("k = " + k);

                System.out.println("m = " + m);
                System.out.println("n = " + n);
                int suma = m + n;
                System.out.println("m + n = " + suma);
                int resta = m - n;
                System.out.println("m - n = " + resta);
                int multiplicacion = m * n;
                System.out.println("m * n = " + multiplicacion);
                int division = m / n;
                System.out.println("m / n = " + division);
        }
}
```

La expresión `++c` indica que se debe incrementar la variable c a su siguiente valor. Cuando una variable es incrementada a su siguiente valor después del valor mayor, se le asigna el valor negativo menor.

Preguntas de repaso

- ¿ Cuantos años tiene el lenguaje Java ?
- ¿ Qué compañía desarrolló el lenguaje Java ?
- ¿ Qué es el código fuente y de donde viene ?
- ¿ Qué tipo de archivos contienen código fuente Java ?
- ¿ Qué es el código de bytes (bytecode) y de donde viene ?
- ¿ Que tipo de archivos contienen código de bytes ?
- ¿ Porque Java es diferente a otros lenguajes de programación ?
- ¿ Que es una Máquina Virtual de Java ?
- ¿ Que es el JDK ?
- ¿ Cual es la diferencia entre los comentarios estilo C y estilo C++ ?
- ¿ Qué es un objeto “stream” (flujo) ?
- ¿ Qué es una excepción ?
- ¿ Cuáles son las diferencias entre una variable y un objeto ?
- ¿ Cuáles son los ocho tipos de datos primitivos de Java ?
- ¿ Que es un tipo referencia ?

Problemas de programación

- Escriba un programa en Java que asigne a una variable entera n un valor inicial y después despliegue cada uno de los dígitos que componen el número
- Escriba un programa en Java que reciba como entrada del teclado un entero que representa la temperatura en la escala Fahrenheit y después calcule y despliegue el equivalente en grados celcios (C = 5 (F-32) / 9)
- Escriba un programa en Java que reciba como entrada del teclado un entero que representa la temperatura en grados celcios y después calcule y despliegue el equivalente en grados Fahrenheit (F = 1.8 C + 32)

2. Las cadenas de caracteres

Una cadena es una secuencia de caracteres. Las palabras, oraciones y nombres son cadenas. En esta unidad se describen las dos clases de cadenas fundamentales: `String` y `StringBuffer`.

2.1. La clase `String`

El tipo mas simple de cadena en Java es un objeto de la clase `String`. Estos objetos son inmutables, no pueden ser cambiados.

Ejemplo 2.1: Operaciones sobre una cadena

```
public class Alfabeto {
      public static void main(String[] args) {
            String alfabeto = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ";
            System.out.println(alfabeto);
            System.out.println("Esta cadena contiene " +
                                alfabeto.length() + "caracteres");
            System.out.println("El caracter en la posicion 4 es " +
                                alfabeto.charAt(4));
            System.out.println("La posicion del caracter Z es " +
                                alfabeto.indexOf('Z'));
            System.out.println("El codigo hash para esta cadena es " +
                                alfabeto.hashCode());
      }
}
```

El objeto llamado alfabeto es declarado en la tercera línea como una instancia de la clase `String` y es inicializado con la literal "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ". El resto del programa consiste de cinco enunciados de salida.

```
ABCDEFGHIJKLMNÑOPQRSTUVWXYZ
Esta cadena contiene 27 caracteres
El caracter en la posicion 4 es E
La posicion del caracter Z es 26
El codigo hash para esta cadena es -1127252723
```

2.2. Utilizando la clase `String`

2.2.1. Las subcadenas

Una subcadena es una cadena cuyos caracteres forman una parte de otra cadena. La clase `String` incluye un método para extraer subcadenas.

Ejemplo 2.2: Subcadenas

```
public class Subcadenas {
      public static void main(String[] args) {
            String alfabeto = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ";
            System.out.println(alfabeto);
            System.out.println("La subcadena en la posición 4 a 7 es " +
                                    alfabeto.substring(4,8));
            System.out.println("La subcadena en la posición 0 a 7 es " +
                                    alfabeto.substring(0,8));
            System.out.println("La subcadena en la posición 8 hasta el
                                    final es " + alfabeto.substring(8));

      }
}
```

La salida sería:

```
ABCDEFGHIJKLMNÑOPQRSTUVWXYZ
La subcadena de la posicion 4 a la 7 es EFGH
La subcadena de la posicion 0 a la 7 es ABCDEFGH
La subcadena de la posicion 8 hasta el final es IJKLMNÑOPQRSTUVWXYZ
```

2.2.2. Cambio de tipo de letras

En Java se distingue entre las mayúsculas y las minúsculas. La clase `String` incluye métodos para cambiar los tipos de letras.

Ejemplo 2.3: Minúsculas y mayúsculas

```
public class MayusculasMinusculas {
      public static void main(String[] args) {
            String mm = "minusculasMAYUSCULAS";
            System.out.println(mm);
            String minusculas = mm.toLowerCase();
            System.out.println(minusculas);
            String mayusculas = mm.toUpperCase();
            System.out.println(mayusculas);
      }
}
```

La salida sería:

```
minusculasMAYUSCULAS
minusculasmayusculas
MINUSCULASMAYUSCULAS
```

2.2.3. Localización de un caracter en una cadena

En la clase `String` se incluyen métodos para encontrar la posición de un caracter dentro de una cadena.

Ejemplo 2.4: Búsqueda de caracteres en una cadena

```
public class BusquedaDeCaracteres {
      public static void main(String[] args) {
            String cadena = "ingenieria en computacion";
            System.out.println(cadena);
            int i = cadena.indexOf('i');
            System.out.println("La primera posicion de 'i' es " + i);
            int j = cadena.indexOf('i',i+1);
            System.out.println("La siguiente posicion de 'i' es " + j);
            int k = cadena.lastIndexOf('i');
            System.out.println("La ultima posición de 'i' es " + k);
      }
}
```

La salida sería:

```
ingenieria en computacion
La primera posicion de 'i' es 0
La siguiente posicion de 'i' es 5
La ultima posicion de 'i' es 22
```

2.2.4. Reemplazo de caracteres en una cadena

En la clase `String` se incluye un método para reemplazar cada ocurrencia de un carácter con otro dentro de una cadena.

Ejemplo 2.5: Reemplazo de caracteres en una cadena

```
public class ReemplazoDeCaracteres {
      public static void main(String[] args) {
            String cadena = "ingenieria en computacion";
            System.out.println(cadena);
            System.out.println(cadena.replace('i','I');
            System.out.println(cadena.replace('c','C');
            System.out.println(cadena);
      }
}
```

La salida sería:

```
ingenieria en computacion
IngenIerIa en computacIon
ingenieria en computacion
```

2.2.5. Conversiones entre tipos primitivos y cadenas

En la clase `String` se incluyen métodos para convertir un tipo primitivo en cadena y en la clase `Integer` se encuentra un método para convertir una cadena en entero.

Ejemplo 2.6: Conversión de tipos primitivos en cadena

```
public class ConversionPrimitivosCadena {
      public static void main(String[] args) {
            boolean b = true;
            char c = '$';
            int n = 44;
            double x = 3.1415926535897932;
            System.out.println("b = " + b);
            System.out.println("c = " + c);
            System.out.println("b = " + n);
            System.out.println("b = " + x);
            String cadenab = String.valueOf(b);
            String cadenac = String.valueOf(c);
            String cadenan = String.valueOf(n);
            String cadenax = String.valueOf(x);
            System.out.println("cadenab = " + cadenab);
            System.out.println("cadenab = " + cadenac);
            System.out.println("cadenab = " + cadenan);
            System.out.println("cadenab = " + cadenax);
      }
}
```

La salida sería:

```
b = true
c = $
n = 44
x = 3.1415926535897932
cadenab = true
cadenac = $
cadenan = 44
cadenas = 3.1415926535897932
```

Ejemplo 2.7: Conversión de cadenas en enteros

```
public class ConversionCadenaEntero {
      public static void main(String[] args) {
            String hoy = "Septiembre 03, 2001";
            String diaCadena = hoy.substring(11,12);
            int diaEntero = Integer.parseInt(diaCadena);
            System.out.println("La fecha de hoy es: " + hoy);
            System.out.println("El dia de hoy es " + diaEntero);
      }
}
```

La salida sería:

```
La fecha de hoy es: Septiembre 03, 2001
El dia de hoy es 3
```

2.3. Los métodos de la clase `String`

A continuación se presenta un resumen de los métodos de la clase `String`.

```
String cadena = new String();
String cadena = new String(char[]);
String cadena = new String(StringBuffer);
String cadena = new String(String);
String cadena = new String(char[],int, int);
String cadena = String.copyValue(char[]);
String cadena = String.copyValue(char[],int,int);
String cadena = String.valueOf(boolean);
String cadena = String.valueOf(char);
String cadena = String.valueOf(int);
String cadena = String.valueOf(float);
String cadena = String.valueOf(double);
String cadena = String.valueOf(char[]);
String cadena = String.valueOf(char[],int,int);
String cadena = String.valueOf(Object);
s = cadena.toString();
i = cadena.length();
c = cadena.charAt(int);
i = cadena.compareTo(String);
s = cadena.concat(String);
b = cadena.endsWith(String);
b = cadena.startsWith(String);
b = cadena.equals(String);
b = cadena.equalsIgnoreCase(String);
i = cadena.hash();
i = cadena.indexOf(char);
i = cadena.indexOf(char,int);
i = cadena.indexOf(String);
i = cadena.indexOf(String,int);
i = cadena.lastIndexOf(char);
i = cadena.lastIndexOf(char,int);
i = cadena.lastIndexOf(String);
i = cadena.lastIndexOf(String,int);
s = cadena.substring(int);
s = cadena.substring(int,int);
s = cadena.toChar();
s = cadena.toLowerCase();
s = cadena.toUpperCase();
```

2.4. La clase `StringBuffer`

La clase `String` es una de las más útiles en Java. Pero sus instancias (objetos) tienen la restricción de ser inmutables. Cuando se desea modificar un `String`, se tiene que hacer construyendo otro `String`, ya sea implícita o explícitamente. Java provee la clase `StringBuffer` para cadenas que necesitan ser modificadas. La razón para tener dos clases es que, para proveer la flexibilidad para cambiar una cadena, se requiere de más espacio en memoria y complejidad.

Ejemplo 2.8: Usando objetos de tipo `StringBuffer`

```
public class ProbandoStringBuffer {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer(10);
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: " +
                                    sb.capacity());
      }
}
```

La salida sería:

```
sb =
La longitud de sb es: 0
La capacidad de sb es: 10
```

El enunciado en la tercera línea crea un objeto (cadena) `StringBuffer` vacío llamado `sb` concapacidad para 10 caracteres. Aquí se ilustra una característica esencial de los objetos `StringBuffer`: estos pueden tener caracteres sin utilizar, lo cual no ocurre con los `String`.

2.5. Utilizando la clase `StringBuffer`

2.5.1. Modificando los objetos `StringBuffer`

El siguiente programa ilustra la flexibilidad de los objetos `StringBuffer`. Se crea sólo un objeto, `sb`, el cual es modificado varias veces usando el operador de concatenación y el método `append()`.

Ejemplo 2.9: Modificando objetos `StringBuffer`

```
public class ModificandoStringBuffer {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer(10);
            sb.append("Facultad de");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: " + sb.capacity());
            sb.append(" Ciencias Quimicas");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: " + sb.capacity());
            sb.append(" e Ingenieria");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: " + sb.capacity());
      }
}
```

La salida sería:

```
sb = Facultad de
La longitud de sb es: 11
La capacidad de sb es: 22
sb = Facultad de Ciencias Quimicas
La longitud de sb es: 29
La capacidad de sb es: 46
sb = Facultad de Ciencias Químicas e Ingenieria
La longitud de sb es: 42
La capacidad de sb es: 46
```

El objeto `sb` es inicializado vacío con capacidad de 10 caracteres. Después de ejecutar el enunciado `sb.append()`, `sb` aumenta su capacidad si es necesario. La capacidad de `sb` es ajustada automáticamente por el sistema operativo.

2.5.2. Reemplazo de caracteres en un `StringBuffer`

El siguiente programa ilustra como reemplazar los caracteres que componen un objeto `StringBuffer`.

Ejemplo 2.10: Reemplazando los caracteres de un objeto `StringBuffer`

```
public class ReemplazandoStringBuffer {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer();
            sb.append("Ingenieria en Computacion");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: "+sb.capacity());
            sb.setCharAt(15,'K');
            System.out.println("sb = " + sb);
            sb.insert(10,'s');
            System.out.println("sb = " + sb);
      }
}
```

La salida sería:

```
sb = Ingenieria en Computacion
La longitud de sb es: 25
La capacidad de sb es: 34
sb = Ingenieria en Komputacion
sb = Ingenierias en Computación
```

Para modificar el contenido de un objeto `StringBuffer` se requiere utilizar los métodos `setCharAt()` e `insert()`.

2.5.3. Conversión de `StringBuffer` en `String`

El siguiente programa ilustra como convertir un objeto `StringBuffer` en un objeto `String`.

Ejemplo 2.11: Conversión de `StringBuffer` en `String`

```
public class ConversionStringBufferAString {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer("Ingenieria");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: "+sb.capacity());
            String s =  sb.toString();
            System.out.println("s = " + s);
            System.out.println("La longitud de s es: " + s.length());
            sb.append("" +s.substring(0,9) + " en Computacion");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: "+sb.capacity());
            System.out.println("s = " + s);
      }
}
```

La salida sería:

```
sb = Ingenieria
La longitud de sb es: 10
La capacidad de sb es: 26
s = Ingenieria
La longitud de s es: 10
sb = Ingenieria en Computacion
La longitud de sb es: 25
La capacidad de sb es: 26
s = Ingenieria
```

En el programa anterior se utiliza el método `toString()` para crear un objeto `String` a partir de un objeto `StringBuffer()`.

2.5.4. Reestableciendo la longitud e invirtiendo un objeto `StringBuffer`

Ejemplo 2.12: Restablecimiento de la longitud e inversión de un `StringBuffer`.

```
public class ReestableciendoLongitud {
      public static void main(String[] args) {
            StringBuffer sb = new StringBuffer("Ingenieria en
                                                Computacion");
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: "+sb.capacity());
            sb.setLength(10);
            System.out.println("sb = " + sb);
```

```
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: "+sb.capacity());
            sb.reverse();
            System.out.println("sb = " + sb);
            System.out.println("La longitud de sb es: " + sb.length());
            System.out.println("La capacidad de sb es: "+sb.capacity());
      }
}
```

La salida sería:

```
sb = Ingenieria en Computacion
La longitud de sb es: 25
La capacidad de sb es: 41
sb = Ingenieria
La longitud de sb es: 10
La capacidad de sb es: 41
sb = aireinegnI
La longitud de sb es: 10
La capacidad de sb es: 41
```

El método `setLength()` incrementa o decrementa la capacidad de un objeto `StringBuffer`. El método *`reverse()`* invierte el orden de los caracteres dentro del *`StringBuffer`*.

2.6. Los métodos de la clase `StringBuffer`

A continuación se presenta un resumen de los métodos de la clase `StringBuffer`.

```
StringBuffer cadena = new StringBuffer();
StringBuffer cadena = new StringBuffer (int);
StringBuffer cadena = new StringBuffer (String);
s = cadena.toString();
i = cadena.length();
i = cadena.capacity();
c = cadena.charAt(int)
cadena.setCharAt(int, char);
cadena.getChars(int, int, char[], int);
cadena.setlength(int);
cadena.ensureCapacity(int);
cadena.append(boolean);
cadena.append(char);
cadena.append(int);
cadena.append(long);
cadena.append(float);
cadena.append(double);
cadena.append(char[]);
cadena.append(objeto);
cadena.append(String);
cadena.append(char[],int,int);
cadena.insert(int,boolean);
cadena.insert(int,char);
cadena.insert(int,int);
```

```
cadena.insert(int,long);
cadena.insert(int,float);
cadena.insert(int,double);
cadena.insert(int,char[]);
cadena.insert(int,objeto);
cadena.insert(int,String);
cadena.reverse();
```

Preguntas de repaso

- ¿ Cual es la diferencia entre String y StringBuffer ?
- ¿ Cual es la diferencia entre la longitud y la capacidad de un StringBuffer ?

Problemas de programación

- Escriba un programa en Java que lea 12 dígitos como un número de teléfono y extraiga los 2 dígitos del código del país, 3 dígitos del código de área y el número de teléfono local, y que despliegue el número de teléfono de la manera convencional. Por ejemplo:
    - Da un numero de teléfono 526646373449
    - El numero completo es + 52 (664) 637-3449
- Escriba un programa en Java que lea una fecha en formato dd/mm/aa y la despliegue en formato dd de “mes” de “año”.

3. Las estructuras de control de programa

3.1. La selección

3.1.1. El enunciado `if … else`

El enunciado `if ... else` permite la ejecución condicionada. Los enunciados contenidos dentro del if serán ejecutados sólo si la condición es verdadera y en caso contrario se ejecutarán los enunciados contenidos dentro del else. La sintaxis para el enunciado `if` es la siguiente:

```
if (condición) { enunciados }
else { enunciados }
```

Donde la condición es una expresión booleana, una expresión cuyo resultado es un valor de tipo *boolean*.

Ejemplo 3.1: Probando dos números enteros aleatorios para saber cual es el mínimo

```
import java.util.Random;

public class ProbandoAleatorios {
      public static void main(String[] args) {
            Random r = new Random();
            int m = r.nextInt();
            System.out.println("m = " + m);
            int n = r.nextInt();
            System.out.println("n = " + n);

            if ( m < n )
                  System.out.println("El minimo es " + m);
            else
                  System.out.println("El minimo es " + n);
      }
}
```

Una salida sería:

```
m = 1589634066
n = -716919032
El minimo es -716919032
```

El enunciado `if ... else` permite la ejecución condicional basada en dos alternativas. Si se desea tener más de dos alternativas posibles, se puede encadenar una secuencia de enunciados `if ... else`.

El siguiente ejemplo genera un número real aleatorio en el rango de 0 a 1 y después despliega en cual de cuatro intervalos cae.

Ejemplo 3.2: Probando el intervalo de un número real aleatorio

```
import java.util.Random;

public class IntervaloAleatorio {
      public static void main(String[] args) {
            Random r = new Random();
            double d = r.nextDouble();
            System.out.println("d = " + d);

            if ( d < 0.25 )
                  System.out.println("0 <= d < 1/4");
            else if ( d < 0.5 )
                  System.out.println("1/4 <= d < 1/2");
            else if ( d < 0.75 )
                  System.out.println("1/2 <= d < 3/4");
            else
                  System.out.println("3/4 <= d < 1");
      }
}
```

Una salida sería:

```
d = 0.5979526952214973
1/2 <= d < 3/4
```

3.1.1.1. Enunciados `if ... else` anidados

El siguiente ejemplo muestra el uso del enunciado `if ... else` anidado.

Ejemplo 3.3: Determinando el orden de tres números utilizando `if ... else` anidados

```
import java.util.Random;

public class TresNumerosIfAnidados {
      public static void main(String[] args) {
            Random r = new Random();
            float f1 = r.nextFloat();
            System.out.println("f1 = " + f1);
            float f2 = r.nextFloat();
            System.out.println("f2 = " + f2);
            float f3 = r.nextFloat();
            System.out.println("f3 = " + f3);

            if ( f1 < f2 )
                  if ( f2 < f3 )
                        System.out.println("f1 < f2 < f3");
                  else
                        if ( f1 < f3 )
                              System.out.println("f1 < f3 < f2");
                        else
                              System.out.println("f3 < f1 < f2");
            else
```

```
                        if ( f1  < f3 )
                              System.out.println("f2 < f1 < f3");
                        else
                              if (f2 < f3)
                                    System.out.println("f2 < f3 < f1");
                              else
                                    System.out.println("f3 < f2 < f1");
      }
}
```

3.1.1.2. Enunciados `if` paralelos

El siguiente ejemplo produce el mismo resultado que el ejemplo 3.3. La diferencia es que los `if ... else` anidados son sustituidos por lo que se llama `if` paralelos.

Ejemplo 3.4: Determinando el orden de tres números utilizando `if` paralelos

```
import java.util.Random;

public class TresNumerosIfParalelos {
      public static void main(String[] args) {
            Random r = new Random();
            float f1 = r.nextFloat();
            System.out.println("f1 = " + f1);
            float f2 = r.nextFloat();
            System.out.println("f2 = " + f2);
            float f3 = r.nextFloat();
            System.out.println("f3 = " + f3);

            if ( f1 < f2 && f2 < f3 )
                  System.out.println("f1 < f2 < f3");
            if ( f1 < f3 && f3 < f2 )
                  System.out.println("f1 < f3 < f2");
            if ( f2 < f1 && f1 < f3 )
                  System.out.println("f2 < f1 < f3");
            if ( f2 < f3 && f3 < f1 )
                  System.out.println("f2 < f3 < f1");
            if ( f3 < f1 && f1 < f2 )
                  System.out.println("f3 < f1 < f2");
            if ( f3 < f2 && f2 < f1 )
                  System.out.println("f3 < f2 < f1");
      }
}
```

El símbolo *&&* es el operador lógico "y". Por ejemplo, la expresión "f1 < f2 && f2 < f3" debe leerse como "f1 menor que f2 y f2 menor que f3".

3.1.1.3. Los operadores lógicos y relacionales

Los operadores que se pueden utilizar en Java para construir expresiones booleanas son los siguientes:

```
Operadores lógicos:         && (y), || (o), ! (negación)
```

```
Operadores relacionales:    < (menor), > (mayor), == (igual),
                            != diferente), <= (menor o igual),
                            >= (mayor o igual)
```

Al igual que los operadores aritméticos, los operadores lógicos pueden encadenarse, combinando mas de dos expresiones. El siguiente ejemplo muestra como combinar cinco expresiones booleanas.

Ejemplo 3.5: Combinación de expresiones booleanas utilizando operadores lógicos

```
public class OperadoresLogicosEncadenados {
      public static void main(String[] args) {
            final int LONGITUD = 255;
            byte buffer[] = new byte[LONGITUD];
            System.out.println("Teclea tu nombre ");

            try { System.in.read(buffer,0,LONGITUD); }
            catch (Exception e) {}

            String nombre = new String(buffer);
            System.out.println("Hola, " + nombre.trim());
            char c = nombre.charAt(0);
            System.out.println("La primera letra de tu nombre es " + c);

            if ( c=='A' || c=='E' || c=='I' || c=='O' || c=='U')
                  System.out.println("La cual es una vocal");
            else
                  System.out.println("La cual es una consonante");
      }
}
```

Una salida sería:

```
Teclea tu nombre Guillermo
Hola, Guillermo
La primera letra de tu nombre es G
La cual es una consonante
```

3.1.1.4. El orden de evaluación de las expresiones

Cuando se tienen diferentes operadores combinados en una expresión, es importante saber en que orden evaluará el compilador los operadores. La prioridad de precedencia para los operadores lógicos, aritméticos y relacionales es:

```
1)++ (incremento postfijo), -- (decremento postfijo)
2)++ (incremento prefijo), -- (decremento prefijo), !
3)*, /, %
4)+, -
5)<, >, <=, >=
6)==, !=
7)&&
```

```
8)||
```

El tipo primitivo `boolean` es el más simple de todos. Los únicos dos valores que puede tener son `false` o `true`. No obstante lo simples que son, las variables booleanas pueden ayudar a simplificar programas con lógica compleja.

El siguiente ejemplo muestra como calcular las raices de una ecuación cuadrática utilizando la fórmula general:

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

Ejemplo 3.6: Implementación de la fórmula cuadrática

```
import java.util.Random;
import java.lang.Math;

public class FormulaCuadratica {
      public static void main(String[] args) {
            Random r = new Random();
            float a = r.nextFloat();
            float b = r.nextFloat();
            float c = r.nextFloat();
            double d = b*b - 4*a*c;
            boolean lineal = (a == 0);
            boolean constante = (lineal && b == 0);
            boolean trivial = (lineal && constante && c == 0);
            boolean sinSolucion = (lineal && constante && c != 0);
            boolean unica = (lineal && b != 0);
            boolean cuadratica = (!lineal);
            boolean compleja = (cuadratica && d < 0);
            boolean igual = (cuadratica && d == 0);
            boolean distinta = (cuadratica && d > 0);

            System.out.println("Los coeficientes de la funcion f(x) =
                                    a*x^2 + b*x + c son: ");
            System.out.println("\ta = " + a);
            System.out.println("\tb = " + b);
            System.out.println("\tc = " + c);
            System.out.println("La ecuacion f(x) = 0 es: ");

            if (lineal)
                  System.out.print("Lineal ");

            if (trivial)
                  System.out.println("y trivial.");

            if (sinSolucion)
                  System.out.println("sin solucion.");
            if (unica) {
                  double x = -c/b;
                  System.out.println("con solucion unica x = " + x);
            }
```

```
            if (cuadratica)
                  System.out.print("Cuadratica con: ");

            if (compleja) {
                  double re = -b/(2*a);
                  double im = Math.sqrt(-d)/(2*a);
                  System.out.println("Solucion compleja: \n\tx1 = " +
                                    re + " + " + im + "i\n\tx2 = " +
                                    re + " - " + im + "i";
            }

            if (igual) {
                  double x = -b/(2*a);
                  System.out.println("Solucion real x = " + x);
            }

            if (distinta) {
                  double s = Math.sqrt(d);
                  double x1 = (-b+s)/(2*a);
                  double x2 = (-b-s)/(2*a);
                  System.out.println("Solucion real: \n\tx1 = " + x1 +
                                          "\n\tx2 = " + x2);
            }
      }
}
```

3.1.2. El operador condicional

Java incluye un operador ternario especial, llamado *operador condicional*, el cual es util para abreviar enunciados `if ... else` simples. Su sintaxis es:

`(condición ? expresión`$_1$ `: expresión`$_2$`)`

El valor de la operación es `expresión`$_1$ o `expresión`$_2$, dependiendo del valor de la `condición` (falso o verdadero).

Ejemplo 3.7: Utilización del operador condicional

```
import java.util.Random;

public class OperadorCondicional {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            float y = r.nextFloat();

            System.out.println("x = " + x + ", y = " + y);
            float min = (x < y ? x : y);
            float max = (x > y ? x : y);
            System.out.println("Minimo = " + min);
            System.out.println("Maximo = " + max);
      }
}
```

3.1.3. El operador de asignación

El operador de asignación es representado por el signo igual "=". Su sintaxis es:

```
var = exp;
```

Esta operación evalua la expresión `exp` y después la asigna a la variable `var`. Por ejemplo:

```
int m;
m = 44;
```

Si `op` es un operador binario, `exp` una expresión y `var` una variable, entonces

```
var op= exp;
```

tiene el mismo efecto que

```
var = var op exp;
```

Por ejemplo:

`n += 7;` es igual que `n = n + 7;`

3.1.4. El enunciado `switch`

El enunciado `switch` es similar a la combinación `if ... else if` que se utiliza para procesar un conjunto de alternativas. La sintaxis del enunciado `switch` es:

```
switch (expresión) {
      case constante1:      enunciados
                            break;
      case constante2:      enunciados
                            break;
      . . .
      case constanteN:      enunciados
                            break;
      default:              enunciados
}
```

Ejemplo 3.8: Utilización del enunciado `switch`

```
import java.util.Random;

public class Switch {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            System.out.println("x = " + x);
```

```
            int calificacion = Math.round(50*x+50);
            System.out.println("Tu calificacion es " + calificacion);
            switch (calificacion/10) {
                  case 10:
                  case 9:     System.out.println("Excelente");
                              break;
                  case 8:     System.out.println("Bueno.");
                              break;
                  case 7:     System.out.println("Se puede mejorar");
                              break;
                  case 6:     System.out.println("Nos vemos despues de
                              clase")
                              break;
                  default:    System.out.println("Muy mal");
            }
      }
}
```

Una salida sería:

```
x = 0.75739926
Tu calificacion es 88
Bueno
```

3.2. La iteración

3.2.1. El enunciado `for`

La sintaxis para el enunciado `for` es:

```
for (expresión1; expresión2; expresión3) {
      enunciados
}
```

Donde $expresión_1$, $expresión_2$ y $expresión_3$ son expresiones boolenas y `enunciados` es un bloque de enunciados. Las tres expresiones sirven para controlar la iteración de los enunciados en el siguiente orden:

1. Evaluar $expresión_1$
2. Evaluar $expresión_2$, si es falsa, salir del ciclo
3. Ejecutar el bloque de enunciados
4. Evaluar $expresión_3$
5. Evaluar la $expresión_2$, si es verdadera, volver al paso 3

Normalmente, las tres expresiones son coordinadas por medio de una variable de control llamada índice o contador, la cual lleva la cuenta de las iteraciones. La estructura común de un enunciado `for` es:

```
for (int i = inicio; i < fin; i++) {
      enunciado1;
      enunciado2;
```

```
            . . .
            enunciado_n;
      }
```

Donde `i` es la variable índice, `inicio` es el primer valor y `fin - 1` es el último valor.

El siguiente ejemplo muestra la función $f(x) = x^2 + x + 41$, la función que utilizó Babage para mostrar su máquina de diferencias.

Ejemplo 3.9: La función de Babbage.

```
public class Babbage {
      public static void main(String[] args) {
            for (int x = 0; x < 10; x++) {
                  int y = x*x+x+41;
                  System.out.println("\t" + x + "\t" + y);
            }
      }
}
```

La salida sería:

```
041
143
247
353
461
571
683
797
8113
9131
```

Los siguientes ejemplos generan un número entre 0 y 100 y prueban si es primo. En el segundo ejemplo se utiliza el enunciado `break` para terminar la iteración.

Ejemplo 3.10: Prueba de números primos

```
import java.util.Random;

public class PrimosFor {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            System.out.println("x = " + x);
            int n = (int) Math.floor(99*x+2);

            for (int d = 0; d < n; d+)
                  if (n%d == 0) {
                        System.out.println(n + " no es un primo");
                        return;
                  }
```

```
            System.out.println(n + " es un primo");
      }
}
```

Ejemplo 3.11: Prueba de números primos utilizando el enunciado `break` para salir la iteración

```
import java.util.Random;

public class PrimosBreak {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            System.out.println("x = " + x);
            int n = (int) Math.floor(99*x+2);
            boolean noEsPrimo = (n < 2);

            for (int d = 0; d < n; d+) {
                  noEsPrimo = (n % 2 == 0);

                  if (noEsPrimo)
                        break;
            }

            if (noEsPrimo)
                  System.out.println(n + " no es un primo");
            else
                  System.out.println(n + " es un primo");
      }
}
```

Para los dos ejemplos, una posible salida sería:

```
x = 0.07461572
7 es un primo
```

3.2.2. El enunciado `while`

El enunciado `for` es la mejor manera de realizar ciclos cuando las repeticiones se encuentran atadas naturalmente a un índice. Cuando no hay una variable de control natural, se recomienda utilizar el enunciado `while`.

La sintaxis para el enunciado `while` es:

```
      while (expresión) {
            enunciados
      }
```

Donde `expresión` es una expresión booleana y `enunciados` es un bloque de enunciados.

El siguiente ejemplo calcula la secuencia de fibonacci. Esta secuencia se define recursivamente de la siguiente manera:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

El proceso en cada iteración es igual: sumar los dos números anteriores. Se llama proceso recursivo porque cada número es recurrente en las dos siguientes ecuaciones.

Ejemplo 3.12: La secuencia de Fibonacci

```
public class Fibonacci {
      public static void main(String[] args) {
            System.out.print(0);
            int fib0 = 0;
            int fib1 = 1;
            int fib2 = fib0 + fib1;

            while (fib2 < 1000) {
                  fib0 = fib1;
                  fib1 = fib2;
                  fib2 = fib0 + fib1;
                  System.out.print(“, “ + fib1);
            }
      }
}
```

La salida sería:

```
0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

El siguiente ejemplo prueba si un número es primo utilizando el enunciado `while`.

Ejemplo 3.13: Prueba de números primos utilizando el enunciado `while`

```
import java.util.Random;

public class PrimosWhile {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            System.out.println(”x = “ + x+);
            int n = (int) Math.floor(99*x+2);
            boolean esPrimo = (n > 1);
            int d = 2;

            while (esPrimo && d < n)
                  esPrimo = (n % d++ != 0);

            if (esPrimo)
                  System.out.println(n + “ es un primo”);
            else
                  System.out.println(n + “ no es un primo”);
      }
}
```

El siguiente ejemplo calcula el máximo común divisor de dos números enteros positivos. Este procedimiento es llamado el algoritmo de Euclides y es el algoritmo no trivial más antiguo que se conoce.

Ejemplo 3.14: El algoritmo de Euclides

```
import java.util.Random;

public class Euclides {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            int m = Math.round(999*x+2);
            x = r.nextFloat();
            int n = Math.round(999*x+2);
            System.out.println("m = " + m + "\t\tn = " + n);

            while (m > 0) {
                  if (m < n) {
                        int temp = m;
                        m = n;
                        n = temp;
                        System.out.println("m = " + m + "\t\tn = " + n);
                  }

                  m -= n;
            }

            System.out.println("El maximo comun divisor es " + n);
      }
}
```

Una salida sería:

```
m = 832     n = 752
m = 752     n = 80
m = 80      n = 32
m = 32      n = 16
El maximo comun divisor es 16
```

3.2.3. El enunciado `do … while`

El enunciado `do ... while` es esencialmente el mismo que el `while`, pero con la condición al final del ciclo.

La sintaxis para el enunciado `while` es:

```
do {
      enunciados
}
while (expresión);
```

Donde `expresión` es una expresión booleana y `enunciados` es un bloque de enunciados.

El siguiente ejemplo calcula el factorial de un número entero entre 0 y 20.

Ejemplo 3.15: Cálculo del factorial

```
import java.util.Random;

public class Factorial {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            int n = Math.round(21*x);
            long f = 1;
            int k = 1;


            do
                  f *= k++;
            while (k <= n);

            System.out.println(n + "! = " + f);
      }
}
```

Una salida sería:

```
5! = 120
```

El siguiente ejemplo prueba si un número es primo utilizando el enunciado `do ... while`.

Ejemplo 3.16: Prueba de primos utilizando el enunciado `do ... while`

```
import java.util.Random;

public class PrimosDoWhile {
      public static void main(String[] args) {
            Random r = new Random();
            float x = r.nextFloat();
            System.out.println("x = " + x+);
            int n = (int) Math.floor(97*x+2);
            boolean esPrimo;
            int d = 2;

            do
                  esPrimo = (n % d++ != 0);
            while (esPrimo && d < n);

            if (esPrimo)
                  System.out.println(n + " es un primo");
            else
                  System.out.println(n + " no es un primo");
```

```
        }
    }
```

3.2.4. Ciclos anidados

Los enunciados dentro de un ciclo pueden ser cualquier tipo de enunciado. Usualmente es un bloque de enunciados y frecuentemente algunos de esos enunciados son ciclos también.

El siguiente ejemplo muestra una tabla de multiplicación.

Ejemplo 3.17: Tabla de multiplicación

```
public class TablaMultiplicacion {
      public static void main(String[] args) {
            final int TAMANO = 12;
            for (int x = 1; x <= TAMANO; x++) {
                  for (y = 1; y <= TAMANO; y++) {
                        int z = x * y;

                        if (z < 10)
                              System.out.println(" ");
                        if (z < 100)
                              System.out.println(" ");

                        System.out.println(" " + z);
                  }

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

La salida sería:

```
1     2     3     4     5     6     7     8     9     10    11    12
2     4     6     8     10    12    14    16    18    20    22    24
3     6     9     12    15    18    21    24    27    30    33    36
4     8     12    16    20    24    28    32    36    40    44    48
5     10    15    20    25    30    35    40    45    50    55    60
6     12    18    24    30    36    42    48    54    60    66    72
7     14    21    28    35    42    49    56    63    70    77    84
8     16    24    32    40    48    56    64    72    80    88    96
9     18    27    36    45    54    63    72    81    90    99    108
10    20    30    40    50    60    70    80    90    100   110   120
11    22    33    44    55    66    77    88    99    110   121   132
12    24    36    48    60    72    84    96    108   120   132   144
```

Preguntas de repaso

- ¿ Porque la combinación `if ... else if` es más general que el enunciado `switch` ?
- ¿ Qué es una condición de continuación ?
- ¿ Para que sirve el enunciado break ?

Problemas de programación

- Escriba un programa en Java que lea un entero entre 2 y 100 y determine si el entero es un número primo
- Escriba un programa en Java que genere 10 enteros aleatorios y los muestre ordenados ascendentemente
- Escriba un programa que lea un mes y dia y que calcule los dias transcurridos desde el primero de enero
- Escriba un programa en Java que desliegue una tabla de la función seno (usando el método `Math.sin()`) en el rango de 0 a Π
- Escriba un programa en Java que calcule la Σi para i =1 hasta n
- Escriba un programa en Java que calcule la Σi^2 para i =1 hasta n

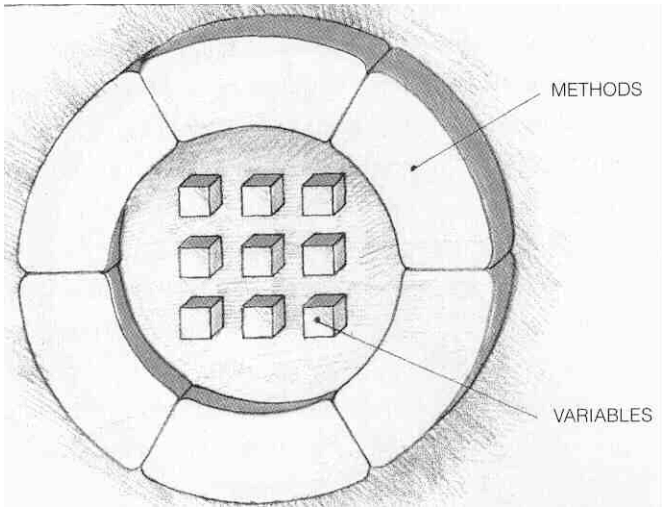
4. Conceptos de orientación a objetos

4.1. Introducción

Se dice que la programación orientada a objetos es más natural que la programación tradicional (imperativa). Esto es cierto en dos niveles. En el primer nivel, la programación orientada a objetos es más natural porque nos permite organizar información de una manera familiar. En un nivel más profundo, es más natural debido a que refleja las técnicas utilizadas por la naturaleza para manejar la complejidad.

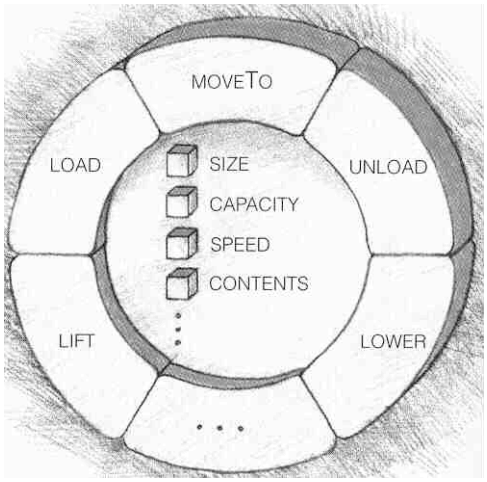
4.2. Los objetos

Un **objeto** es un paquete de software que contiene una colección de métodos y campos (también llamados atributos).

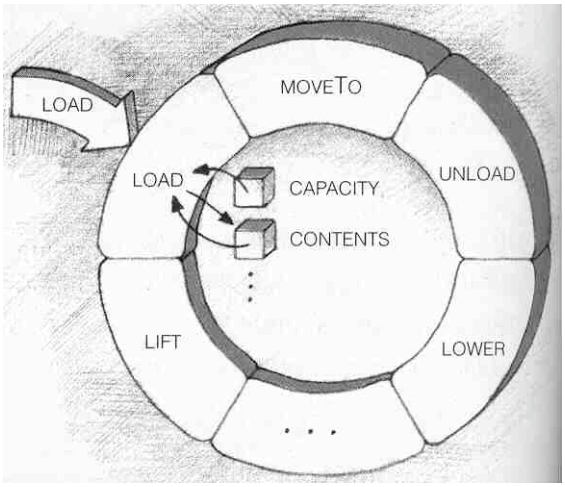


La acción de empaquetar juntos datos y procedimientos es llamada **encapsulación**. Los mecanismos de encapsulación de la tecnología orientada a objetos es una extensión de la estrategia de ocultamiento de información desarrollada en la programación estructurada. La tecnología orientada a objetos mejora esta estrategia utilizando mejores mecanismos para juntar la información correcta y ocultar sus detalles de manera efectiva.

Consideremos el siguiente objeto “montacarga” con algunos de sus campos y métodos.



En el enfoque orientado a objetos, los datos dentro de un objeto son accesibles sólo por los métodos del objeto.

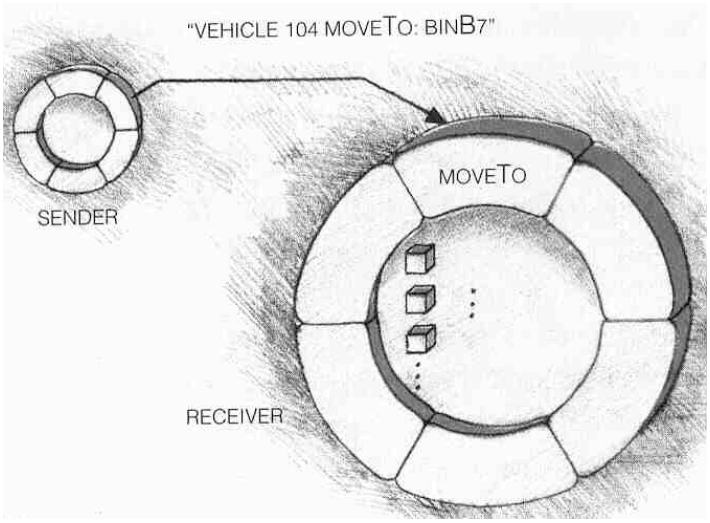


4.3. Los mensajes

Los objetos interactúan a través de mensajes que invocan a los métodos. Un **mensaje** consiste del nombre del objeto seguido del nombre del método que se desea ejecutar. Si un método requiere más información, se deben incluir **parámetros**.

La comunicación basada en mensajes ofrece dos tipos de protección:

- Protege los campos de un objeto de ser corrompidos por otros objetos. Si los objetos tuvieran acceso directo a los campos, eventualmente alguno de ellos podría manejar de manera incorrecta algún campo dañando al objeto.
- Al ocultar sus campos, un objeto protege a otros objetos de las complicaciones de su estructura interna.

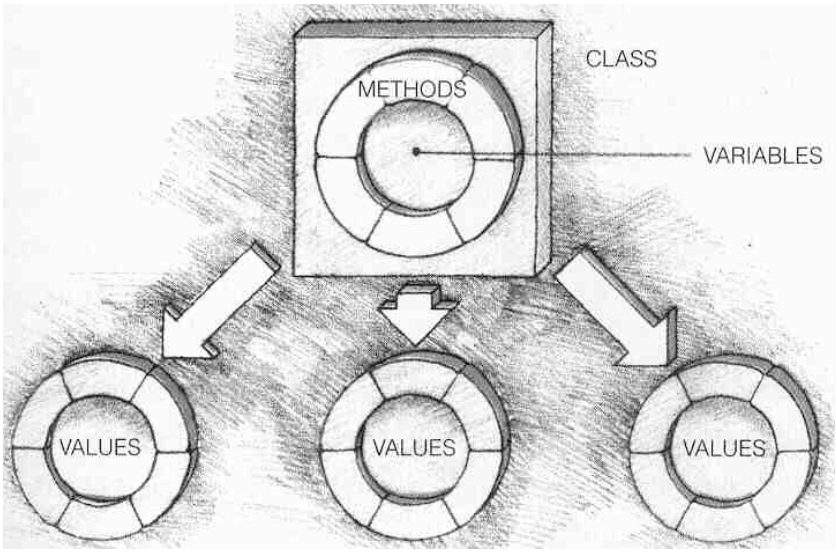


Un programa en la programación orientada a objetos consiste de un número de objetos interactuando a través de mensajes.

4.4. Las clases

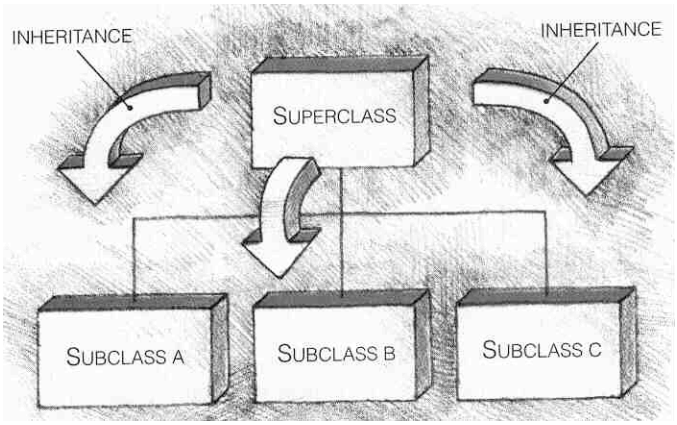
Algunas veces un programa involucra sólo un objeto. Sin embargo es más común que sea necesario contar con más de un objeto de cada tipo

Una **clase** es una plantilla que define los métodos y atributos que estarán incluidos en un tipo particular de objeto. Las descripciones de los métodos y los campos son incluidas sólo una vez en la definición de la clase. Los objetos que pertenecen a una clase, llamados instancias de la clase, contienen los valores particulares para los campos.

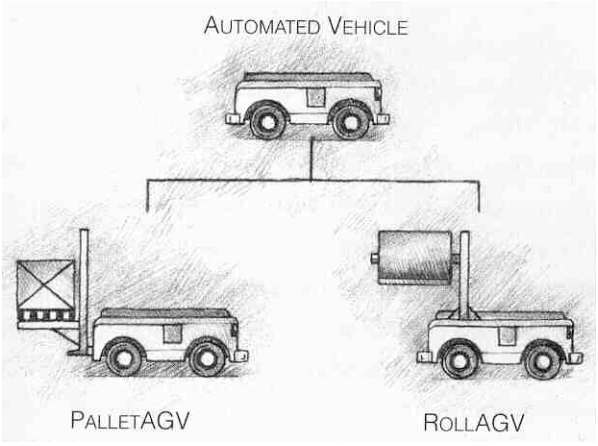


4.5. La herencia

La **herencia** es un mecanismo que permite definir una clase de objetos como un caso especial de una clase más general, incluyendo automáticamente los métodos y campos de la clase general. Los casos especiales de una clase son conocidos como **subclases** de esa clase y la clase más general se conoce como **superclase** de sus casos especiales. Además de los métodos y atributos que hereda, la subclase puede definir los suyos propios y redefinir cualquiera de las características heredadas.



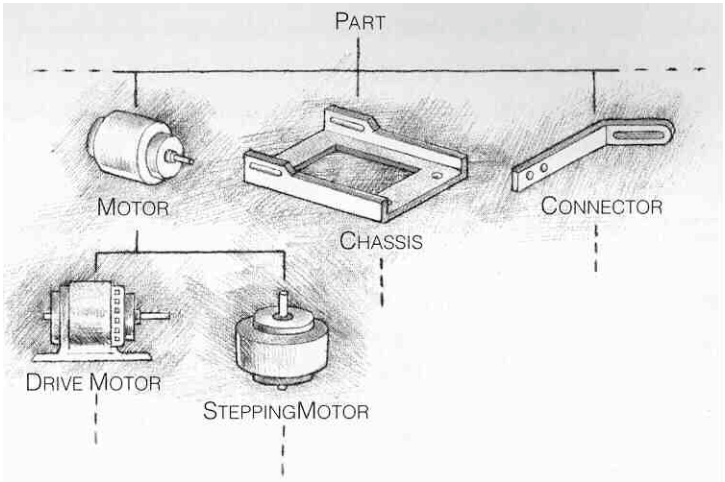
Por ejemplo la clase que representa al montacargas (vehículo de carga automático) puede ser dividida en dos subclases: *PalletAGV* (con paleta) y *RollAGV* (con rodillo), cada una de las cuales hereda las características generales de la clase padre.



4.6. Las jerarquías de clases

Las clases pueden ser clasificadas en muchos niveles y la herencia automáticamente se va acumulando a través de los niveles. La estructura de árbol resultante se conoce como **jerarquía de clases**. Por ejemplo, una clase llamada parte (*part*) puede ser dividida en tipos

especiales de esa parte como motor, chasis, conector (*connector*), etc. El motor a su vez puede dividirse en otros dos tipos de motor.



La invención de la jerarquía de clases es la parte más importante de la tecnología orientada a objetos. El conocimiento humano es estructurado exactamente de esta manera. Descansa en conceptos generales y su refinamiento en casos especializados. La tecnología orientada a objetos usa el mismo mecanismo conceptual que nosotros utilizamos en la vida diaria para construir software complejo pero fácil de entender.

4.7. Los lenguajes de programación orientados a objetos

Existe un gran número de lenguajes de programación orientados a objetos, todos ellos implementan los conceptos cubiertos anteriormente. Algunos de los lenguajes más populares son:

Simula
* Diseñado en 1967 por Ole-Johan Dahl y Krysten Nygaard
* Es considerado como el primer lenguaje orientado a objetos

Smalltalk
* Desarrollado en el PARC de XEROX a principios de los setenta
* Es un lenguaje orientado a objetos puro

C++
* Diseñado por Bjarne Stroustrup de AT&T en 1985
* Extiende C con varias características de orientación a objetos

Eiffel
* Diseñado por Bertrand Meyer en 1988
* Es un lenguaje orientado a objetos puro construido cuidadosamente
* Utiliza una sintaxis familiar a la de los lenguajes convencionales

5. Los métodos

5.1. ¿ Que es un método y como se define ?

Un método es una secuencia de declaraciones y enunciados ejecutables encapsulados como un mini programa independiente. En otros lenguajes de programación a los métodos se les llama funciones, procedimientos, subrutinas o subprogramas. En Java cada enunciado ejecutable debe estar dentro de algún método. Los programadores diseñan los programas decidiendo primero que acciones específicas se tienen que realizar y que objetos las realizarán.

5.2. Un par de ejemplos sencillos

El siguiente ejemplo prueba un método llamado `cubo()` que regresa el cubo de un entero que es pasado como argumento.

Ejemplo 5.1: El método `cubo()`

```
public class PruebaCubo {
      public static void main(String[] args) {
            for (int i=0; i<6; i++)
                  System.out.println(i + "\t" + cubo(i));
      }

      public static int cubo(int n) {
            return n*n*n;
      }
}
```

La salida sería:

```
00
11
28
327
464
5125
```

El siguiente programa prueba un método llamado `min()` que regresa el mínimo de dos valores enteros.

Ejemplo 5.2: El método `min()`

```
import java.util.Random;

public class Pruebamin {
      public static void main(String[] args) {
            Random r = new Random();
```

```
            for (int i=0; i<5; i++) {
                  float x = r.nextFloat();
                  int m = Math.round(100*x);
                  x = r.nextFloat();
                  int n = Math.round(100*x);
                  int y = min(m,n);
                  System.out.println("min(" + m + "," + n + ") = " + y);
            }
      }

      public static int min(int x, int y) {
            if (x < y)
                  return x;
            else
                  return y;
      }
}
```

Una salida sería:

```
min(16, 18) = 16
min(83, 30) = 30
min(68, 96) = 68
min(17, 73) = 17
min(72, 26) = 26
```

5.3. Las variables locales

Una variable local es una variable que se declara dentro de un método. Estas variables sólo pueden usarse dentro de ese método y dejan de existir cuando el método finaliza. En el ejemplo 5.1, la variable `i` es local al método `main()` y `n` es local a `cubo()`. En el ejemplo 5..2, las variables `r, i, x, m, n, y` son locales a `main()` y los parámetros `x, y` son locales a `min()`.

4.4. Métodos que se invocan a si mismos

En los ejemplos anteriores hemos visto como un método puede invocar a otro método, el método `main()` invoca a los métodos `min()` y `cubo()`. Ahora veremos métodos que se invocan a si mismos, estos métodos son llamados *recursivos*.

Algunos procesos son naturalmente recursivos. Por ejemplo la función factorial se puede definir en forma recursiva como se muestra a continuación:

$$n! = \begin{cases} 1, \text{ si } n = 0 \\ n\,(n-1)!, \text{ si } n > 0 \end{cases}$$

Ejemplo 5.3: Una implementación recursiva de la función factorial

```
public class PruebaFactorial {
      public static void main(String[] args) {
            for (int i=0; i<9; i++)
```

```
                System.out.println("f(" + i +") = " + factorial(i));

            try { System.in.read(); }
            catch (Exception e) {}
        }

        public static long factorial(int n) {
            if (n < 2)
                return 1;
            else
                return n*factorial(n-1);
        }
}
```

Una salida sería:

```
f(0) = 1
f(1) = 1
f(2) = 2
f(3) = 6
f(4) = 24
f(5) = 120
f(6) = 720
f(7) = 5040
f(8) = 40320
```

Una definición recursiva tiene dos partes esenciales: la base, que define la función para los primeros valores, y su relación recursiva, que define el n-ésimo valor en términos de valores previos.

4.5. Métodos booleanos

Un método booleano es un método que regresa un tipo `boolean`. Estos métodos son invocados como expresiones booleanas para controlar ciclos y condicionales.

Ejemplo 5.4: El método `esPrimo()`

```
public class PruebaPrimos {
    public static void main(String[] args) {
        for (int i=0; i<80; i++)
            if (esPrimo(i))
                System.out.println(i + " ");

    }

    public static boolean esPrimo(int n) {
        if (n < 2)
            return false;
        if (n == 2)
            return true;
        if (n % 2 == 0)
            return false;
```

```
            for (int d=3; d<Math.sqrt(n); d+=2)
                  if (n % d == 0)
                        return false;

            return true;
      }
}
```

La salida sería:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 49 53 59 61 67 71 73 79
```

5.6. Método nulos

Un método nulo es un método que regresa un tipo `void`. Esto quiere decir que el método no regresa ningún valor.

Ejemplo 5.5: El método `esBisiesto()`

```
public class PruebaBisiesto {
      public static void main(String[] args) {
            prueba(1492);
            prueba(1810);
            prueba(1999);
            prueba(2000);
            prueba(2001);
      }

      public static boolean esBisiesto(int n) {
            if (n < 1582)
                  return false;

            if (n % 400 == 0)
                  return true;

            if (n % 100 == 0)
                  return false;

            if (n % 4 == 0)
                  return true;

            return false;
      }

      public static void prueba(int n) {
            if (esBisiesto(n))
                  System.out.println(n + " es año bisiesto");
            else
                  System.out.println(n + " no es año bisiesto");
      }
}
```

La salida sería:

```
1492 no es año bisiesto
1810 no es año bisiesto
1999 no es año bisiesto
2000 es año bisiesto
2001 no es año bisiesto
```

5.7. Sobrecargado de nombres de métodos

En Java es posible utilizar el mismo nombre para diferentes métodos, siempre que tengan diferente lista de tipos de parámetros. Esto se conoce como sobrecargado.

Ejemplo 5.6: Los métodos `max()`

```
import java.util.Random;

public class PruebaMaximo {
      public static void main(String[] args) {
            Random r = new Random();

            for (int i=0; i<5; i++) {
                  float x = r.nextFloat();
                  int a = Math.round(100*x);
                  x = r.nextFloat();
                  int b = Math.round(100*x);
                  x = r.nextFloat();
                  int c = Math.round(100*x);
                  System.out.println("max(" + a + "," + b + "," +
                                          c + ") = " + max(a,b,c));
            }
      }

      public static int max(int m, int n) {
            if (m > n)
                  return m;
            else
                  return n;
      }

      public static int max(int n1, int n2, int n3) {
            return max(max(n1,n2),n3);
      }
}
```

La salida sería:

```
max(34,43,19) = 43
max(11,36,65) = 65
max(8,40,46) = 46
max(67,44,4) = 67
max(58,48,19) = 58
```

Preguntas de repaso

- ¿ Qué es una variable local ?
- ¿ Qué es un método recursivo ?
- ¿ Cuales son las dos partes de una definición recursiva ?
- ¿ Qué es un método booleano ?
- ¿ Qué es un método nulo ?
- ¿ Qué es el sobrecargado ?

Problemas de programación

- Escriba un programa que invoque los siguientes métodos en forma recursiva:
    - Máximo común divisor
    - Potencia
- Escriba y pruebe un programa que contenga un método recursivo para calcular el n-ésimo número triangular. t(n) = t(n-1) + n, para n > 1
- Escriba y pruebe un método booleano que determine si un número es triangular

6. Las clases

6.1. ¿ Que es una clase ?

Un programa es una colección de uno o más archivos de texto que contienen clases escritas en Java, al menos una de las cuales es pública (`public`) y contiene un método llamado `main()` que tiene la siguiente forma:

```
public static void main(String[] args) {
      // enunciados del método
}
```

Una clase de Java es una categoría específica de objetos, similar a un tipo Java (como el `int`, `char`, etc.). Una clase de Java especifica el rango de valores que sus objetos pueden tener. Los valores que puede tener un objeto se llama su estado.

Las clases de Java tienen tres características esenciales que no tienen los tipos de datos primitivos:

1. Las clases pueden ser definidas por el programador
2. Los objetos de las clases pueden contener variables, incluyendo referencias a otros objetos
3. Las clases pueden contener métodos que le dan a sus objetos la habilidad de actuar

La programación orientada a objetos significa escribir programas que definan clases cuyos métodos lleven a cabo las instrucciones del programa. Los programas se diseñan decidiendo que objetos serán usados y que acciones realizarán.

Supongamos que se requiere escribir un programa para geometría plana. En este caso, los objetos son muy obvios: puntos, lineas, triángulos, círculos, etc. Podemos abstraer los elementos geométricos definiendo clases cuyos objetos los representen. En el ejemplo 6.1 se presenta una clase que representa un punto en el plano cartesiano.

Ejemplo 6.1: La clase punto

```
public class Punto {
      private double x, y;

      public Punto(double a, double b) {
            x = a;
            y = b;
      }

      public double getX() {
            return x;
      }

      public double getY() {
            return y;
```

```
        }

        public boolean equals(Punto p) {
              return (x == p.x && y == p.y);
        }

        public String toString() {
              return new String("(" + x + ", " + y + ")");
        }
}

public class PruebaPunto {
        public static void main(String[] args) {
              Punto p = new Punto(2,3);
              System.out.println("p.getX()= " + p.getX() +
                                 ", p.getY()= " + p.getY());
              System.out.println("p = " + p);

              Punto q = new Punto(7,4);
              System.out.println("q = " + q);

              if (p.equals(q))
                    System.out.println("p es igual a q");
              else
                    System.out.println("p no es igual a q");
        }
}
```

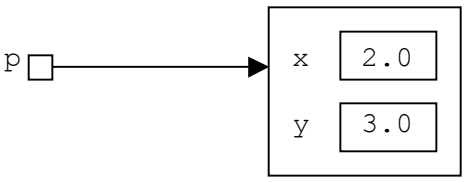
La salida sería:

```
p.getX()= 2, p.getY()=3
p = (2, 3)
q = (7, 4)
p no es igual a q
```

La clase tiene dos campos `x, y` cuyos valores son las coordenadas del punto que representa el objeto. La primera línea en el método `main()` es

```
Punto p = new Punto(2,3);
```

Esto hace tres cosas. Primero, declara p como una referencia a objetos de la clase `Punto`. Después aplica el operador `new` para crear un objeto `Punto` con valores 2, 3 para los campos `x, y`. Finalmente inicializa la referencia `p` con el nuevo objeto.



La segunda línea en `main()` invoca a los métodos `getX(), getY()` para obtener los valores de los campos `x, y`.

La tercera línea invoca al método `toString()`. Este es un método especial que es invocado al momento de pasar un objeto al método `println()`.

En la sexta línea se invoca al método `equals()` para determinar si dos objetos de tipo `Punto` son iguales.

6.2. Declaración de clases

El propósito de una declaración es introducir un identificador al compilador. La declaración provee toda la información que requiere el compilador para compilar los enunciados que incluyan al identificador que se está declarando. En Java todas las clases, campos, variables locales, constructores y métodos deben declararse.

La sintaxis para una declaración de clase es:

```
modificadores class nombre-de-la-clase {
      cuerpo
}
```

Donde los `modificadores` pueden ser algunas de las palabras reservadas { `public, abstract, final` }, `nombre-de-la-clase` es cualquier identificador válido y `cuerpo` es una secuencia de declaraciones de campos, constructores y métodos.

La sintaxis para los campos es la siguiente:

```
modificadores tipo-de-dato nombre-del-campo;
```

Donde los `modificadores` pueden ser algunas de las palabras reservadas { `static, final, transient, volatile, public protected, private` }, el `tipo-de-dato` es alguno de los ocho tipos predefinidos de Java o una clase y el `nombre-de-la-variable` es cualquier identificador válido.

La sintaxis para un constructor es la siguiente:

```
modificador nombre-de-la-clase(lista-de-parametros) {
      cuerpo
}
```

Donde el `modificador` puede ser alguna de las palabras reservadas { `public, protected, private` }, el *`nombre-de-la-clase`* es el nombre de la clase donde se está declarando el método, `lista-de-parametros` es una secuencia de declaraciones de parámetros y `cuerpo` es una secuencia de declaraciones de enunciados que serán ejecutados por el constructor al momento de crear el objeto.

Ejemplo 6.2: La clase `Linea`

```
public class Linea {
      private Punto p0;   // un punto sobre la linea
      private double m;  // la pendiente de la linea

      public Linea(Punto p, double s) {
            p0 = p;
```

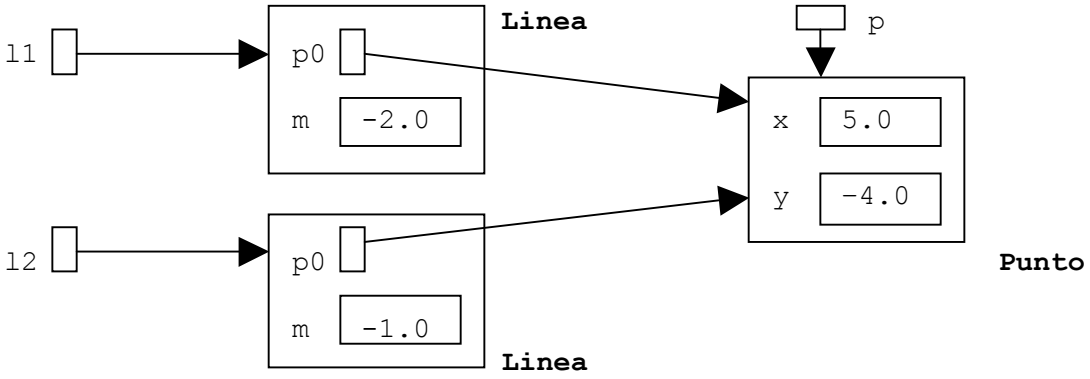
```
            m = s;
      }

      public double getPendiente() {
            return m;
      }


      public double interseccionY() {
            return (p0.y()-m*p0.x());
      }

      public boolean equals(Linea l) {
            return (getPendiente() == l.getPendiente() &&
                        interseccionY() == l.interseccionY());
      }

      public String toString() {
            return new String("y = " + (float)m + "x + " +
                                    (float)interseccionY());
      }
}

public class PruebaLinea {
      public static void main(String[] args) {
            Punto p = new Punto(5,-4);
            Linea l1 = new Linea(p,-2);
            System.out.println("La linea 1 es " + l1);
            System.out.println("Su pendiente es " + l1.getPendiente() +
                                    ", su interseccion con y es " +
                                    l1.interseccionY());
            Linea l2 = new Linea(p,-1);
            System.out.println("La linea 2 es " + l2);
            System.out.println("Su pendiente es " + l2.getPendiente() +
                                    ", su interseccion con y es " +
                                    l2.interseccionY());

            if (l1.equals(l2))
                  System.out.println("Las lineas son iguales");
            else
                  System.out.println("Las lineas no son iguales");
      }
}
```

La salida sería:

```
La linea 1 es y = -2.0x + 6.0
Su pendiente es -2.0, su interseccion con y es 6.0
La linea 2 es y = -1.0x + 1.0
Su pendiente es -1.0, su interseccion con y es 1.0
Las lineas no son iguales
```

Este programa empieza creando dos objetos: `p` de tipo `Punto` y `l1` de tipo `Linea`. `p` tiene dos campos de tipo `double x, y` con valores 5, -4. `l1` también tiene dos campos `p0` de tipo `Punto` y `m` de tipo double. `p0` es una referencia a `p` y `m` tiene el valor –2. Después de

crear los objetos `p` y `l1`, el programa prueba los métodos `toString()`, `getPendiente()` e `interseccionY()`.



6.3. Los modificadores

Las siguientes tablas resumen los modificadores que pueden aparecer en las declaraciones de clases, variables de clase, variables locales, constructores y métodos.

Modificadores de clase	
Modificador	**Significado**
public	Está accesible desde todas las clases
abstract	La clase no puede ser instanciada
final	No se pueden declarar subclases

Modificadores de constructores	
Modificador	**Significado**
public	Está accesible desde todas las clases
protected	Está accesible sólo dentro de su clase y sus subclases
private	Está accesible sólo dentro de su clase

Modificadores de campos	
Modificador	**Significado**
public	Está accesible desde todas las clases
protected	Está accesible sólo dentro de su clase y sus subclases
private	Está accesible sólo dentro de su clase
static	Sólo existe un valor del campo para todas las instancias de las clases
transient	No es parte del estado persitente de un objeto
voaltile	Puede ser modificado por hilos (threads) asíncronos
final	Debe ser inicializado y no puede ser modificado

Modificadores de variables locales	
Modificador	**Significado**
final	Debe ser inicializada y no puede ser modificada

Modificadores de métodos	
Modificador	**Significado**
public	Está accesible desde todas las clases
protected	Está accesible sólo dentro de su clase y sus subclases
private	Está accesible sólo dentro de su clase
abstract	No tiene cuerpo y pertenece a una clase abstracta
final	No puede ser sobreescrita por ninguna clase
static	Está atado a la clase en vez de a una instancia de la clase
native	Su cuerpo es implementado en otro lenguaje de programación
synchronized	Debe ser bloqueado antes que un hilo (thread) lo invoque

Los tres modificadores de acceso `public`, `protected` y `private` se utilizan para especificar donde puede ser usada la entidad declarada. Si no se especifica ningún modificador, entonces la entidad tendrá acceso de paquete, lo cual significa que puede ser accesada desde cualquier clase en el paquete.

El modificador `static` se usa para especificar que un método pertenece a una clase. Sin el modificador, el método sería un método de instancia, el cual sólo puede invocarse cuando está atado a un objeto de la clase. El objeto es llamado el argumento implícito del método. Un método de clase es un método que se puede invocar sin estar atado a ningún objeto específico de la clase.

El modificador final tiene tres significados diferentes, dependiendo de que tipo de entidad modifique. Si modifica una clase, significa que la clase no puede tener sibclases. Si modifica un campo o variable local, significa que la variable debe ser inicializada y no podrá modificarse. Si modifica un método, significa que el método no podrá sobre escribirse en ninguna subclase.

6.4. Los constructores

Las clases pueden tener tres tipos de miembros: campos, métodos y constructores. Un campo es una variable declarada como miembro de la clase. Un método es una función que es usada para realizar alguna acción a través de instancias de la clase. Un constructor es una clase especial de función cuyo único propósito es crear objetos de la clase.

6.4.1. Definición de constructor

Los constructores se diferencian de los métodos de clase en tres cosas:

```
1.Los constructores tienen el mismo nombre que la clase
2.Los constructores no tienen tipo de regreso
3.Los constructores son invocados por el operador new
```

Cada clase tiene al menos un constructor para instanciarla. Si no se declaran constructores para la clase, el compilador declarará automáticamente uno sin argumentos. Este es llamado el constructor por omisión. Sin embargo, es conveniente declarar constructores para controlar la manera en que son creados los objetos.

Ejemplo 6.3: La clase linea con tres constructores

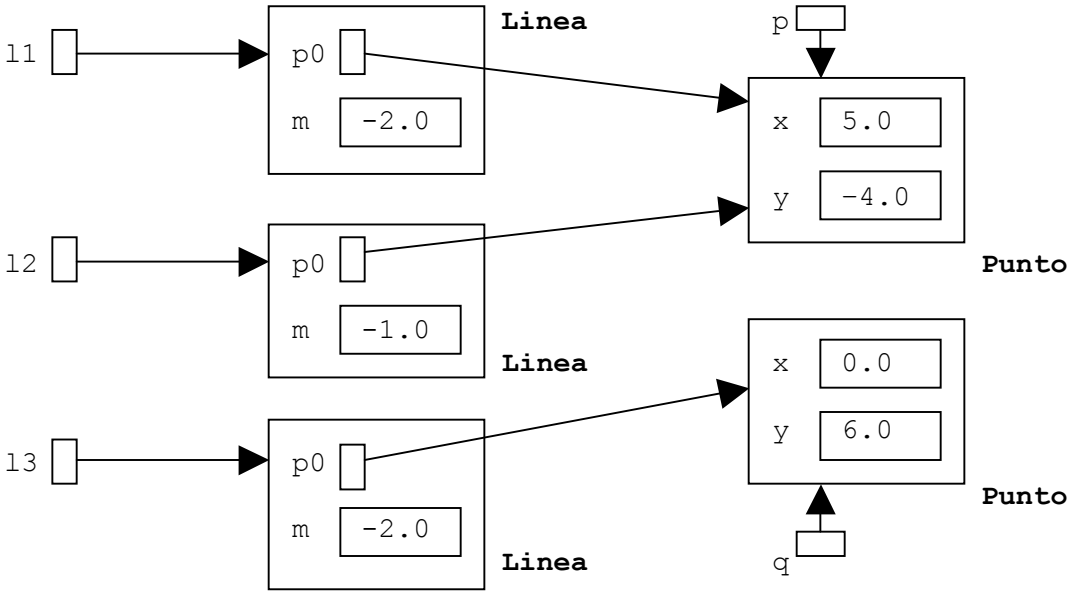
```
public class Linea {
      private Punto p0;   // un punto sobre la linea
      private double m;   // la pendiente de la linea

      public Linea(Punto p, double s) {
            p0 = p;
            m = s;
      }

      public Linea(Punto p, Punto q) {
            p0 = p;
            m = (p.y() - q.y())/(p.x() - q.x());
      }

      public Linea(double a, double b) {
            p0 = new Punto(a,b);
            m = -b/a;
      }

      public double getPendiente() {
            return m;
      }

      public double interseccionY() {
            return (p0.y()-m*p0.x());
      }

      public boolean equals(Linea l) {
            return (getPendiente() == l.getPendiente() &&
                        interseccionY() == l.interseccionY());
      }

      public String toString() {
            return new String("y = " + (float)m + "x + " +
                                    (float)interseccionY());
      }
}

public class PruebaLinea {
      public static void main(String[] args) {
            Punto p = new Punto(5,-4), q = new Punto(-1,2);
            Linea l1 = new Linea(p,-2);
            Linea l2 = new Linea(p,q);
            Linea l3 = new Line(3,6);

            System.out.println("La linea 1 es " + l1);
```

```
            System.out.println("La linea 2 es " + l2);
            System.out.println("La linea 3 es " + l3);

            if (l1.equals(l3))
                  System.out.println("Las lineas son iguales");
            else
                  System.out.println("Las lineas no son iguales");
      }
}
```

La salida sería:

```
La linea 1 es y = -2.0x + 6.0
La linea 2 es y = -1.0x + 1.0
La linea 3 es y = -2.0x + 6.0
Las lineas son iguales
```

Este programa crea cinco objetos: `p, q` de tipo `Punto` y `l1, l2, l3` de tipo `Linea`. Se utilizan los métodos `toString()` para desplegar en la pantalla los valores de los campos de los objetos Linea y el método `equals()` para probar si una línea es igual a otra. En este ejemplo cada objeto de tipo `Linea` `l1, l2, l3` es construido a través de un constructor diferente.



6.4.2. Constructores copia (copy constructors)

El constructor más simple que puede tener una clase es uno que no tiene parámetros. Este es llamado el constructor por omisión (default constructor). Otro tipo simple de constructor es aquel cuyo parámetro es una referencia a un objeto de la misma clase a la cual pertenece el constructor. Esta forma es usada para duplicar un objeto existente, es llamado constructor copia (copy constructor).

Ejemplo 6.4: Duplicando un objeto `Punto`

```
public class Punto {
      private double x, y;

      public Punto(double a, double b) {
            x = a;
            y = b;
      }

      public Punto(Punto p) {
            x = p.x;
            y = p.y
      }

      public double getX() {
            return x;
      }

      public double getY() {
            return y;
      }

      public boolean equals(Punto p) {
            return (x == p.x && y == p.y);
      }

      public String toString() {
            return new String("(" + x + ", " + y + ")");
      }
}

public class PruebaPunto {
      public static void main(String[] args) {
            Punto p = new Punto(2,3);
            System.out.println("p = " + p);
            Punto q = new Punto(p);
            System.out.println("q = " + q);

            if (p.equals(q))
                  System.out.println("p contiene lo mismo que q");
            else
                  System.out.println("p no contiene lo mismo que q");

            if (p == q)
                  System.out.println("y p == q");
            else
                  System.out.println(", pero p != q");
      }
}
```

La salida sería:

```
p = (2.0, 3.0)
q = (2.0, 3.0)
p contiene lo mismo que q, pero p != q
```

Los objetos `p` y `q` contienen los mismos valores para los campos `x, y`, debido a que se utiliza el constructor copia de `p` para crear `q`.

6.4.3. Constructores por omisión (default constructors)

En Java cada campo de una clase es inicializado automáticamente a su valor inicial para su tipo. La tabla siguiente muestra los valores para los tipos predefinidos y los tipos referencia.

Valores iniciales para los campos de una clase	
Tipo	**Valor inicial**
boolean	false
char	'\u0000'
integer	0
floating point	0.0
reference	null

La siguiente clase ilustra el uso de un constructor por omisión declarado implícitamente.

Ejemplo 6.5: Una clase que representa un monedero

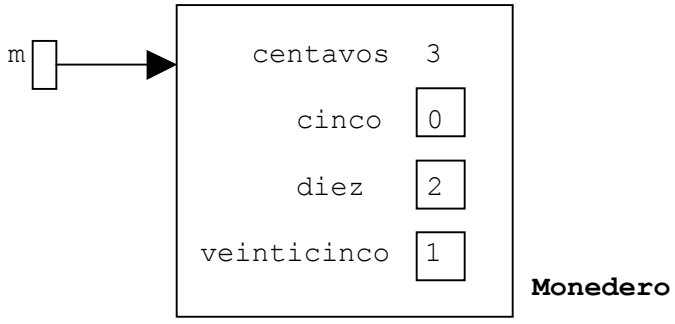
```
public class Monedero {
      private int centavos;
      private int cinco;
      private int diez;
      private int veinticinco;

      public float dolares() {
            int c = centavos + 5*cinco + 10*diez + 25*veinticinco;
            return (float) c/100;
      }

      public void insertar(int c, int cc, int dc, int vc) {
            centavos += c;
            cinco += cc;
            diez += dc;
            veinticinco += vc;
      }

      public void remover(int c, int cc, int dc, int vc) {
            centavos -= c;
            cinco -= cc;
            diez -= dc;
            veinticinco -= vc;
      }

      public String toString() {
            return new String(veinticinco + " de veinticinco + " +
```

```
                                        diez + " de diez + " + cinco +
                                        " de cinco + " + centavos +
                                        " centavos = " + dolares());
        }
}

public class PruebaMonedero {
        public static void main(String[] args) {
                Monedero m = new Monedero();

                System.out.println(m);
                m.insertar(3,0,2,1);
                System.out.println(m);
                m.insertar(3,1,1,3);
                System.out.println(m);
                m.remover(3,1,0,2);
                System.out.println(m);
                m.remover(0,0,0,4);
                System.out.println(m);
        }
}
```

La salida sería:

```
0 de veinticinco + 0 de diez + 0 de cinco + 0 centavos = 0.0
1 de veinticinco + 2 de diez + 0 de cinco + 3 centavos = 0.48
4 de veinticinco + 3 de diez + 1 de cinco + 6 centavos = 1.41
2 de veinticinco + 3 de diez + 0 de cinco + 3 centavos = 0.83
-2 de veinticinco + 3 de diez + 0 de cinco + 3 centavos = -0.17
```

En la primera línea se declara una referencia a objetos de tipo `Monedero` y después se invoca al constructor por omisión para crear un objeto `Monedero` vacío. Posteriormente se utilizan los métodos `insertar()`, `println()` y `remover()` para visualizar el funcionamiento del monedero.



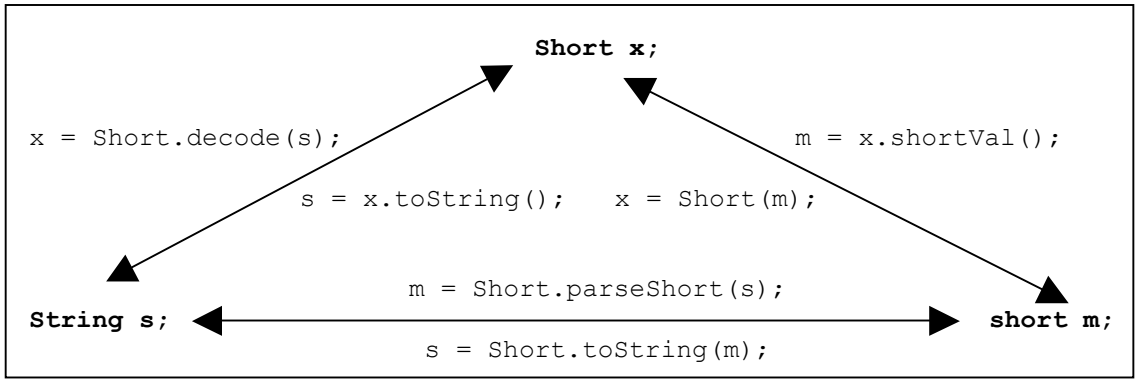
La clase `Monedero` tiene una falla, diferentes estados pueden resultar en la misma cantidad en dolares, ademas de permitirse remover monedas que no están disponibles. Esto no ocurre con los tipos primitivos. Para hacer que las clases se comporten más como tipos primitivos, es importante asegurar que los objetos tienen una representación única. Esto puede lograrse especificando y forzando los invariantes en las clases.

Un *invariante de clase* es una condición impuesta sobre los campos de todos los objetos de una clase. El objetivo más común de un invariante de clase es garantizar la representación única.

6.5. Clases de envoltura (wrapper classes)

Cada uno de los ocho tipos de datos primitivos de Java tienen una clase correspondiente llamada clase de envoltura (wrapper class) que generalizan el tipo. Estas clases de envoltura están definidas en el paquete `java.lang` y pueden ser usadas sin utilizar el enunciado `import`. Los nombres de las clases de envoltura son: `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` y `Double`. Se les llama clases de envoltura porque encapsulan tipos predefinidos de tal forma que una variable puede ser representada por un objeto cuando sea necesario. También proveen valores máximos y mínimos para el tipo y las dos clases de punto flotante definen las constantes `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` y `NaN`.

El siguiente diagrama muestra los seis métodos para conversión que pueden ser usados para conversiones entre el tipo `short` y la clase `Short` y `String`. Existen métodos similares para las otras siete clases de envoltura.



Al igual que la clase `String`, las clases de envoltura son declaradas finales. Esto significa que sus objetos son sólo de lectura, sus valores no pueden ser modificados.

Preguntas de repaso

- ¿ Qué es el estado de una clase ?
- ¿ Cual es la diferencia entre un campo y una variable local ?
- ¿ Que ventajas se tienen al incluir en una clase, un método `toString()` sin parámetros y que regresa un objeto `String` ?
- ¿ Cual es la diferencia entre un constructor y un método ?
- ¿ Cual es la diferencia entre un método accesador y un método mutador ?
- ¿ Qué es un argumento implícito ?
- ¿ Porque es ilegal que un método estático invoque a uno que no lo es ?
- ¿ Cual es la diferencia entre la igualdad de objetos y la igualdad de las referencias a estos ?
- ¿ Cual es la diferencia entre un miembro público y uno privado en una clase ?

- ¿ Que es un invariante de clase ?
- ¿ Qué es un constructor por omisión ?
- ¿ Que es un constructor copia ?
- ¿ Cual es la diferencia entre invocar un constructor copia y usar una asignación ?

Problemas de programación

- Utilice las clases `Punto` y `Linea` para escribir otras clases que representen: circunferencia, elipse y parábola
- Escriba otra versión de la clase `Punto` definida en base a coordenadas polares en vez de coordenadas rectangulares
- Escriba una clase para representar una persona, a través de los campos: nombre, peso, estatura, fecha de nacimiento, etc. y defina los constructores y métodos necesarios para accesar, modificar e imprimir el estado de los objetos de la clase Persona
- Modifique la clase Monedero agregando invariantes de clase para garantizar que no existan fallas

7. La composición, la herencia y las interfaces

Una de las características de la programación orientada a objetos que la hace tan poderosa es la facilidad con que se puede reutilizar el código para diferentes propósitos. Esto es posible gracias a la composición y herencia.

7.1. La composición

La composición es la creación de una clase usando otra clase como su campo componente.

El siguiente ejemplo muestra una clase que define un nombre de persona.

Ejemplo 7.1: Una clase que representa un nombre

```
public class Nombre {
	private String apellidoPaterno;
	private String apellidoMaterno;
	private String primerNombre;
	private String segundoNombre;

	Nombre(String primerNombre, String apellidoPaterno,
			String apellidoMaterno) {
		this.primerNombre = primerNombre;
		this.apellidoPaterno = apellidoPaterno;
		this.apellidoMaterno = apellidoMaterno;
	}

	public Nombre(String primerNombre, String segundoNombre,
			String apellidoPaterno, String apellidoMaterno) {
		this.primerNombre = primerNombre;
		this.segundoNombre = segundoNombre;
		this.apellidoPaterno = apellidoPaterno;
		this.apellidoMaterno = apellidoMaterno;
	}

	public String getPrimerNombre() {
		return primerNombre;
	}

	public String getSegundoNombre() {
		return segundoNombre;
	}

	public String getApellidoPaterno() {
		return apellidoPaterno;
	}

	public String getApellidoMaterno() {
		return apellidoMaterno;
```

```
	}

	public void setPrimerNombre(String primerNombre) {
		this.primerNombre = primerNombre;
	}

	public void setSegundoNombre(String segundoNombre) {
		this.segundoNombre = segundoNombre;
	}

	public void setApellidoPaterno(String apellidoPaterno) {
		this.apellidoPaterno = apellidoPaterno;
	}

	public void setApellidoMaterno(String apellidoMaterno) {
		this.apellidoMaterno = apellidoMaterno;
	}

	public String toString() {
		String s = new String();

		if (primerNombre != null)
			s += primerNombre + " ";

		if (segundoNombre != null)
			s += segundoNombre + " ";

		if (apellidoPaterno != null)
			s += apellidoPaterno + " ";

		if (apellidoMaterno != null)
			s += apellidoMaterno + " ";

		return s.trim();
	}
}
```

La palabra reservada `this` puede ser usada dentro de un método de instancia para referirse al argumento implícito. Es decir, al objeto al cual está atado este método al ser invocado.

El siguiente ejemplo muestra la clase `Persona`, la cual utiliza objetos de la clase nombre.

Ejemplo 7.2: Una clase que representa una persona

```
Public class Persona {
	protected Nombre nombre;
	protected char sexo;
	protected String id;

	public Persona(Nombre nombre, char sexo) {
		this.nombre = nombre;
		this.sexo = sexo;
	}
```

```
	public Persona(Nombre nombre, char sexo, String id) {
		this.nombre = nombre;
		this.sexo = sexo;
		this.id = id;
	}

	public Nombre getNombre() {
		return nombre;
	}

	public char getSexo() {
		return sexo;
	}

	public String getId() {
		return id;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public void setSexo(char sexo) {
		this.sexo = sexo;
	}

	public void setId(String id) {
		this.id = id;
	}

	public String toString() {
		String s = new String(nombre + " (Sexo: " + sexo);

		if (id != null)
			s += ", Id: " + id;

		s += ")";
		return s;
	}
}

class PruebaPersona {
	public static void main(String[] args) {
		Nombre nombreAviles = new Nombre("Eduardo", "Aviles",
                                                 "Dorantes");
		Persona aviles = new Persona(nombreAviles,'M');
		System.out.println("Aviles: " + aviles);
		aviles.getNombre().setSegundoNombre("Antonio");
		System.out.println("Aviles: " + aviles);
		aviles.setId("2/38760");
		System.out.println("Aviles: " + aviles);
	}
}
```

La salida sería:

```
Aviles: Eduardo Aviles Dorantes (Sexo: M)
```

```
Aviles: Eduardo Antonio Aviles Dorantes (Sexo: M)
Aviles: Eduardo Antonio Aviles Dorantes (Sexo: M, Id: 2/38760)
```

7.2. Clases recursivas

Un método recursivo es uno que se invoca a si mismo como ya vimos en la unidad 4. Una clase recursiva es aquella que está compuesta por ella misma, es decir, contiene campos que se refieren a objetos de la misma clase. Las clases recursivas proveen una técnica poderosa para construir estructuras enlazadas que pueden representar relaciones complejas eficientemente.

El siguiente ejemplo muestra la clase `Persona`, utilizando campos que hacen referencia a personas, en este caso a la madre y el padre.

Ejemplo 7.3: Una clase recursiva que representa una persona

```
public class Persona {
      protected Nombre nombre;
      protected char sexo;
      protected String id;
      protected Persona madre;
      protected Persona padre;
      private static final espacios = "  ";
      private static String sangria = "";

      public Persona(Nombre nombre, char sexo) {
            this.nombre = nombre;
            this.sexo = sexo;
      }

      public Persona(Nombre nombre, char sexo, String id) {
            this.nombre = nombre;
            this.sexo = sexo;
            this.id = id;
      }

      public Nombre getNombre() {
            return nombre;
      }

      public char getSexo() {
            return sexo;
      }

      public String getId() {
            return id;
      }

      public Persona getMadre() {
            return madre;
      }

      public Persona getPadre() {
            return padre;
      }
```

```
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public void setSexo(char sexo) {
		this.sexo = sexo;
	}

	public void setId(String id) {
		this.id = id;
	}

	public void setMadre(Persona madre) {
		this.madre = madre;
	}

	public void setPadre(Persona padre) {
		this.padre = padre;
	}

	public String toString() {
		String s = new String(nombre + " (" + sexo + ")");

		if (id != null)
			s += ", Id: " + id;

		s += "\n";


		if (madre != null) {
			sangria += espacios;
			s += sangria + "Madre: " + madre;
			sangria = sangria.substring(2);
		}

		if (padre != null) {
			sangria += espacios;
			s += sangria + "Padre: " + padre;
			sangria = sangria.substring(2);
		}

		return s;
	}
}

public class PruebaPersona {
	public static void main(String[] args) {
		Persona juan = new Persona(new Nombre("Juan", "Gonzalez",
								"Lopez"),'M');
		Persona maria = new Persona(new Nombre("Maria", "Martinez",
								"Perez"),'F');
		Persona jose = new Persona(new Nombre("Jose", "Luis",
							"Gonzalez", "Martinez"),'M');
		jose.setMadre(maria);
		jose.setPadre(juan);
		System.out.println(jose);
```

```
        }
    }
```

La salida sería:

```
Jose Luis Gonzalez Martinez (M)
  Madre: Maria Martinez Perez (F)
  Padre: Juan Gonzalez Lopez (M)
```

Este programa crea tres objetos de tipo `Persona`. Esta versión de la clase `Persona` es recursiva ya que los campos `madre` y `padre` son referencias a objetos de la misma clase `Persona`.

Ejemplo 7.4: Una lista de números telefónicos

```
public class Amigo {
      protected String nombre;
      protected String telefono;
      protected Amigo siguiente;
      static Amigo lista;;

      public Amigo(String nombre, String telefono) {
            this.nombre = nombre;
            this.telefono = telefono;
            this.siguiente = lista;
            lista = this;
      }

      public static void imprimir() {
            Amigo amigo = lista;
            if (amigo == null)
                  System.out.println("La lista esta vacia");
            else
                  do {
                        System.out.println(amigo);
                        amigo = amigo.siguiente;
                  }
                  while(amigo != null);
      }

      public public String toString() {
            return new String(nombre + ":\t" + telefono);
      }
}

class PruebaAmigo {
      public static void main(String[] args) {
            Amigo.imprimir();
            new Amigo("Jose David Bayliss Lozano","612-3456");
            new Amigo("Carlo Alvertti Chavez Meza","678-9012");
            new Amigo("Mario Humberto Cornejo Manzo","634-5678");
            Amigo.imprimir();
      }
}
```

La salida sería:

```
La lista esta vacia
Mario Humberto Cornejo Manzo: 634-5678
Carlo Alvertti Chavez Meza: 678-9012
Jose David Bayliss Lozano: 612-3456
```

La lista consiste de una secuencia de `Amigos` enlazados. Cada objeto tiene tres campos: `nombre, telefono` y `siguiente`. El campo *siguiente* es una referencia al siguiente objeto en la lista.

El modificador `static` en la declaración de la variable `lista` la identifica como una variable de clase (en vez de una variable de instancia). Esto significa que sólo existe una de esas variables para la clase (en vez de una para cada objeto). Esta variable de clase es una referencia al primer objeto de la lista.

El modificador `static` en el método `print()` lo identifica como un método de clase. Este método despliega la lista completa o indica si está vacía. Funciona siguiendo la liga `siguiente` en los objetos.

7.3. La herencia

La herencia consiste en la creación de una clase (subclase) a través de la extensión de otra clase (superclase), de tal forma que los objetos de la subclase heredan automáticamente los campos y métodos de su superclase.

Ejemplo 7.5: La subclase `Y` hereda el campo `m` de la clase `X`

```
public class X {
      protected int m;

      public String toString() {
            return new String("( " + m + ")");
      }
}

public class Y extends X {
      private int n;

      public String toString() {
            return new String("( " + m + ", " + n + ")");
      }
}

public class PruebaY {
      public static void main(String[] args) {
            X x = new X();
            System.out.println("x = " + x);
            Y y = new Y();
            System.out.println("y = " + y);
      }
}
```

La salida sería:

```
x = (0)
y = (0, 0)
```

El campo `m` debe ser declarado `protected` en vez de `private`, ya que si se declara `private` sólo estará accesible dentro de su clase y al declararlo `protected` se permite el acceso a cualquiera de sus subclases.

7.4. Sobreescritura de campos y métodos

Un objeto `y` de la clase `Y` es esencialmente igual a un objeto `x` de la clase `X` con más datos y funcionalidad. La diferencia se da cuando, por ejemplo, ambas clases declaran un método `g()`. El enunciado `y.g()` invocará al método declarado en la clase `Y`, no al declarado en la clase `X`. En este caso decimos que el método `g()` ha sido sobreescrito.

La sobreescritura de campos es similar a la sobreescritura de métodos: tienen la misma declaración, pero en diferentes clases.

Ejemplo 7.6: Sobreescritura de campos y métodos

```
public class X {
      protected int m;
      protected int n;

      public void f() {
            System.out.println("Metodo f() de la clase X");
            m = 22;
      }

      public void g() {
            System.out.println("Metodo g de la clase X");
            n = 22;
      }

      public public String toString() {
            return new String("( " + m + ", " + n + ")");
      }
}

public class Y extends X {
      private double n;

      public void g() {
            System.out.println("Metodo g de la clase Y");
            n = 3.1415926535897932;
      }

      public String toString() {
            return new String("( " + m + ", " + n + ")");
      }
}
```

```
public class PruebaY {
      public static void main(String[] args) {
            X x = new X();
            x.f();
            x.g();
            System.out.println("x = " + x);

            Y y = new Y();
            y.f();
            y.g();
            System.out.println("y = " + y);
      }
}
```

7.5. La palabra reservada `super`

Java usa la palabra reservada `super` para referirse a miembros de la superclase. Cuando se usa como `super()`, se invoca el constructor de la superclase. Cuando se usa como `super.f()`, se invoca al método `f()` declarado en la superclase. Esto permite tener acceso a los métodos que han sido sobreescritos.

Ejemplo 7.7: La clase `Estudiante` subclase de `Persona`

```
public class Estudiante extends Persona {
      protected int creditos;
      protected double promedio;

      public Estudiante(Nombre nombre,char sexo,int creditos,double promedio)
      {
            super(nombre,sexo);
            this.creditos = creditos;
            this.promedio = promedio;
      }

      public int getCreditos() {
            return creditos;
      }

      public double getPromedio() {
            return promedio;
      }

      public void setCreditos(int c) {
            creditos = c;
      }

      public void setPromedio(double p) {
            promedio = p;
      }

      public String toString() {
            String s;
```

```
            s = new String(super.toString());
            s += "\n\tCreditos: " + creditos;
            s += "\n\tPromedio: " + promedio;
            return s;
      }
}

public class PruebaEstudiante {
      public static void main(String[] args) {
            Nombre nombreCota = new Nombre("Carlos", "Antonio", "Cota",
                                                "Apodaca");
            Estudiante cota = new Estudiante(nombreCota,"M', 56, 9.7);
            System.out.println("Cota: " + cota);
      }
}
```

La salida sería:

```
Cota: Carlos Antonio Cota Apodaca (Sexo: M)
      Creditos: 56
      Promedio: 9.7
```

La primera linea del constructor de la clase `Estudiante` es `super(nombre,sexo)`, lo cual invoca al constructor de la superclase de `Estudiante` que es `Persona`.

7.6. Herencia contra composición

Herencia significa especialización. Una subclase especializa sus campos y métodos a través de la herencia y puede agregar más según se requiera. Al agregar campos y métodos se hace a la clase más restrictiva, más especial.

Mientras que la herencia significa especialización, la composición significa agregación. La clase Estudiante es una especialización de la clase Persona, mientras que es un agregado de las clases Nombre y String.

7.7. Las Jerarquías de clases

Una clase puede tener más de una subclase y las subclases pueden tener a su vez subclases. Estas relaciones nos llevan naturalmente a tener una árbol de clases. En una jerarquía o árbol de clases, decimos que la clase Y es un descendiente de la clase X si hay una secuencia de clases que empieza con Y y termina con X en la cual cada clase es la superclase de la clase anterior. También podemos decir que X es un ancestro de Y.

Dentro de una jerarquía de clases, hay dos tipos especiales de clases: clases abstractas y clases finales, las cuales son identificadas por los modificadores `abstract` y `final`.

Una clase abstracta es una clase que tiene al menos un método abstracto. Un método abstracto es un método que se declara sólo con su firma, no tiene implementación. Dado que una clase abstracta tiene por lo menos un método abstracto, no se pueden crear objetos de esta.

El siguiente ejemplo define tres clases: la clase abstracta `Figura` y las dos clases (concretas) `Circulo` y `Cuadrado`.

Ejemplo 7.8: Una clase abstracta

```
public abstract class Figura {
      public abstract Punto getCentro();
      public abstract double diametro();
      public abstract double area();
}

public class Circulo extends Figura {
      private Punto centro;
      private double radio;

      public Circulo(Punto centro, double radio) {
            this.centro = centro;
            this.radio = radio;
      }

      public Punto getCentro() {
            return centro;
      }

      public double getRadio() {
            return radio;
      }

      public double diametro() {
            return 2*radio;
      }

      public double area() {
            return Math.PI*radio*radio;
      }

      public String toString() {
            return new String("{centro = " + centro + ", radio = " +
                                    radio " "}");
      }
}

public class Cuadrado extends Figura {
      private Punto esquinaSuperiorIzquierda;
      private double lado;

      public Cuadrado(Punto esquinaSuperiorIzquierda, double lado) {
            this.esquinaSuperiorIzquierda = esquinaSuperiorIzquierda;
            this.lado = lado;
      }

      public Punto getCentro() {
            Punto c = new Punto(esquinaSuperiorIzquierda);
            c.trasladar(lado/2, -lado/2);
            return c;
      }
```

```
	public double diametro() {
		return lado*Math.sqrt(2.0);
	}

	public double area() {
		return lado*lado;
	}

	public String toString() {
		return new String("{esquina superior izquierda = " +
					esquinaSuperiorIzquierda +
					", lado = " + lado + "}");
	}
}

public class PruebaCirculo {
	public static void main(String[] args) {
		Circulo circulo = new Circulo(new Punto(3,1), 2.0);
		System.out.println("El circulo es: " + circulo);
		System.out.println("Su centro es: " + circulo.getCentro());
		System.out.println("Su diametro es: " + circulo.diametro());
		System.out.println("Su area es: " + circulo.area());
	}
}
```

La salida sería:

```
El circulo es: {centro = (3.0, 1.0), radio = 2.0}
Su centro es: (3.0, 1.0)
Su diametro es: 4.0
Su area es: 12.566370614359172
```

```
public class PruebaCuadrado {
	public static void main(String[] args) {
		Cuadrado cuadrado = new Cuadrado(new Punto(1,5), 3.0);
		System.out.println("El cuadrado es " + cuadrado);
		System.out.println("Su centro es: " + cuadrado.getCentro());
		System.out.println("Su diametro es: "+ cuadrado.diametro());
		System.out.println("Su area es: " + cuadrado.area());
	}
}
```

La salida sería:

```
El cuadrado es: {esquina superior izquierda = (1.0, 5.0), lado = 3.0}
Su centro es: (2.5, 3.5)
Su diametro es: 4.242640687119286
Su area es: 9.0
```

Un método abstracto puede ser visto como la especificación de un contrato. En él se especifica que es lo que deben implementar sus subclases, pero deja la implementación a ellas. Un método abstracto es aquel que se pretende sea sobreescrito en cada una de las clases de la familia de subclases. Un método final es lo opuesto: un método que no puede

ser sobreescrito en ninguna subclase. La razón para declarar un método final es para garantizar que no será modificado.

Una clase abstracta es aquella que tiene por lo menos un método abstracto. De la misma manera, una clase final es aquella que tiene por lo menos un método final.

7.8. La clase `object` y la jerarquía de clases de Java

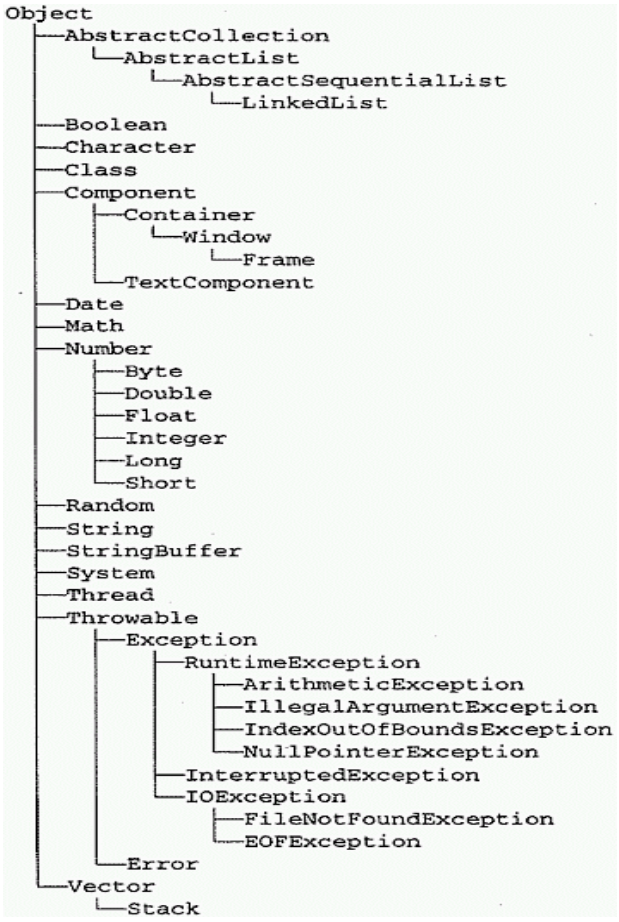
Java define una clase especial llamada `Object`, la cual es el ancestro de cualquier otra clase. En la clase `Object` se declaran doce miembros: un constructor y once métodos. Dado que cada clase es subclase de `Object`, cada objeto puede invocar estos métodos. Cuatro de los métodos, `clone()`, `hashCode()`, `equals()` y `toString()` están para ser sobreescritos.

Si no se usa explicitamente la palabra reservada extends para hacer la nueva clase una subclase de otra, automaticamente será subclase de `Object`. Por ejemplo las dos definiciones siguientes son equivalentes.

```
class Punto {
     double x, y;
}

class Punto extends Object {
     double x, y;
}
```

El diagrama de la derecha muestra una pequeña parte (menos del 3%) de la jerarquía de clases en Java. Podemos ver varias de las clases que hemos utilizado anteriormente son subclases de `Object` como: `String`, `Math`, `Random`, `System`. Notemos también que las seis clases de envoltura para los tipos numéricos son sublclases de `Number`.

Las subclases de `Throwable` son las clases utilizadas para manejar los errores en tiempo de ejecución. La clase `Vector` encapsula las características de los arreglos que serán vistos en la unidad 8. Las subclases de Component son las clases utilizadas para construir interfaces de usuario gráficas.



7.9. Los métodos `clone()` y `equals()`

Los métodos `clone()` y `equals()` son dos de los doce métodos declarados en la clase `Object`. Son declarados ahí para forzarnos a sobreescribirlos en nuestras clases, con la finalidad de facilitar el duplicado y comparación de objetos.

Ejemplo 7.9: Un método `equals()` para la clase `Punto`

```
class Punto {
      double x, y;

      public Punto(double a, double b) {
            x = a;
            y = b;
      }

      public boolean equals(Punto p) {
            return (x == p.x && y == p.y);
      }

      public String toString() {
            return new String("(" + x + ", " + y + ")");
      }
}
```

El método `equals()` definido anteriormente funciona, pero no sobreescribe el método `Object.equals()`, porque su firma es diferente. A continuación se muestra el método que sobreescribiría `Object.equals()`.

```
public boolean equals(Object p) {
      return (x == p.x && y == p.y);
}
```

Ejemplo 7.10: La clase `Punto` con mejores definiciones de los métodos `equals()` y `clone()`

```
public class Punto {
      double x, y;

      public Punto(double a, double b) {
            x = a;
            y = b;
      }

      public public Object clone() {
            return new Punto(x,y);
      }

      public boolean equals(Object p) {
            if (p instanceof Punto)
                  return (x == ((Punto)p).x && y == ((Punto)p).y);
            else
```

```
                return false;
        }

        public String toString() {
                return new String("(" + x + ", " + y + ")");
        }
}

public class PruebaPunto {
        public static void main(String[] args) {
                Punto p = new Punto(2,3);
                System.out.println("p = " + p);
                Punto q = (Punto) p.clone();
                System.out.println("q = " + q);

                if (p == q)
                        System.out.println("p == q");
                else
                        System.out.println("p != q");

                if (p.equals(q))
                        System.out.println("p es igual a q");
                else
                        System.out.println("p no es igual a q");
        }
}
```

La salida sería:

```
p = (2.0, 3.0)
q = (2.0, 3.0)
p != q
p igual a q
```

En este ejemplo tenemos dos objetos `Punto` con los mismos datos, uno de ellos producto de clonar al otro. Por lo tanto el operador de comparación `==` los encuentra diferentes, mientras que el método `equals()` los encuentra iguales. La forma en que se sobreescribieron los métodos `clone()` y `equals()` los hace son consistentes con los métodos que aparecen en las biliotecas de clases estandar de Java.

7.10. Las interfaces

7.10.1. ¿ Que es una interfaz ?

Una interfaz es una clase abstracta que no contiene ningún detalle de implementación. La sintaxis para definir una interfaz en Java es la siguiente:

```
public inteface Nombre-interfaz {
        Cuerpo-interfaz
}
```

La interfaz tiene la apariencia de una declaración de clase sólo que utiliza la palabra reservada `interface`. Consiste de una lista de métodos que deben ser posteriormente implementados.

Se dice que una clase implementa una interfaz si define todos los métodos abstractos de la interfaz. Para utilizar una interfaz se utiliza la palabra reservada `implements`.

```
class Nombre-clase extends Nombre-superclase implements Nombre-interfaz
{
      Cuerpo-clase
}
```

Ejemplo 7.11: Una interfaz para `Figura`

```
public interface Figura {
      Punto centro();
      public double diametro();
      public double area();
}

class Circulo implements Figura {
      private Punto centro;
      private double radio;

      public Circulo(Punto centro, double radio) {
            this.centro = centro;
            this.radio = radio;
      }

      public Punto centro() {
            return centro;
      }

      public double diametro() {
            return 2*radio;
      }

      public double area() {
            return Math.PI*radio*radio;
      }

      public String toString() {
            return new String("{centro = " + centro +
                                    ", radio = " + radio " "}");
      }
}

class PruebaCirculo {
      public static void main(String[] args) {
            Circulo circulo = new Circulo(new Punto(3,1), 2.0);
            System.out.println("El circulo es: " + circulo);
            System.out.println("Su centro es: " + circulo.centro());
            System.out.println("Su diametro es: " + circulo.diametro());
            System.out.println("Su area es: " + circulo.area());
      }
}
```

La salida sería:

```
El circulo es: {centro = (3.0, 1.0), radio = 2.0}
Su centro es: (3.0, 1.0)
Su diametro es: 4.0
Su area es: 12.566370614359172
```

En este ejemplo, la interfaz `Figura` es equivalente a la clase abstracta `Figura` definida en el ejemplo 7.8.

7.10.2. Diferencias entre interfaz y clase

Aun cuando una interfaz puede verse como algo equivalente a una clase abstracta, existen ciertas diferencias:

1) Una clase abstracta puede definir algunos métodos abstractos y otros concretos, mientras que en una interfaz todos sus métodos son implícitamente abstractos
2) En una clase abstracta puede haber todo tipo de variables, mientras que en una interfaz sólo puede haber variables estáticas, finales, es decir, constantes

7.10.3. Jerarquías de clases entre interfaces

Las interfaces pueden tener subinterfaces, tal como las clases tienen subclases. Una subinterfaz hereda todos los métodos abstractos y constantes de la superinterfaz, y puede definir nuevos métodos abstractos y constantes. La diferencia entre la herencia de clases y la herencia de interfaces es que las interfaces pueden heredar de más de una interfaz a la vez.

```
interface Nombre-interfaz extends interfaz1, interfaz2, interfaz3, ... {
      Cuerpo-interfaz
}
```

Ejemplo 7.12: La interfaz `Figura` heredando de dos interfaces

```
public interface EstiloDeLinea {
      final int normal = 0, grueso = 1;
      final int negro = 0, rojo = 1, verde = 2, azul = 3, blanco = 4;

      public void tipoLinea(int tl);
}

public interface EstiloDeRelleno {
      final boolean sinRelleno = false, relleno = true;
      final int normal = 0, lineasVerticales = 1, lineasHorizontales = 2,
                  cuadriculado = 3;

      public void relleno(boolean r, int p);
}
```

```
public interface Figura extends EstiloDeLinea, EstiloDeRelleno {
      public Punto centro();
      public double diametro();
      public double area();
}
```

Preguntas de repaso

- ¿ Cual es la diferencia entre composición y herencia ?
- En el método `imprimir()` en el ejemplo 6.4, ¿ Porque es necesario usar una variable local `amigo`, en vez del campo `lista` directamente ?
- ¿ Cual es la diferencia entre sobrecargado y sobreescritura de métodos ?
- ¿ Que es polimorfismo ?
- ¿ Que es una interfaz en Java ?
- ¿ Cual es la diferencia entre interfaces y clases ?

Problemas de programación

- Implemente clases para representar lo siguiente:
  - Una dirección postal
  - Un número telefónico
  - Una dirección de correo electrónico
- Modifique la clase `Persona` agregando los siguientes campos:
  - `protected Direccion direccion;`
  - `protected Telefono telefono;`
  - `protected Email email;`
- Modifique el método `insertar()` del ejemplo 6.4 para que los objetos sean agregados en la lista en orden alfabético
- Escriba una interfaz llamada *Comparable* que obligue a la clase que la implemente a incluir un método para comparar objetos de su clase

8. Arreglos y vectores

8.1. ¿ Que es un arreglo ?

Un arreglo es un objeto que consiste de una secuencia de elementos numerados que tienen el mismo tipo. Los elementos son numerados a partir del 0 y pueden ser referidos por su número usando el operador `[]`.

8.2. Arreglos de caracteres

Uno de los tipos de arreglos más simples son aquellos cuyos elementos son de tipo `char`. Ya se vio en la unidad 2 que los objetos `String` son parecidos a los arreglos de caracteres.

Ejemplo 8.1: Comparando arreglos de caracteres y objetos `String`

```
class PruebaArregloCaracteres {
      public static void main(String [] args) {
            String s = new String("ABCDEFG");
            char[] a = s.toCharArray();

            System.out.println("s = " + s + "\ta = " + a);
            System.out.println("s.length() = " + s.lengt() +
                                    "\ta.length() = " + a.length);

            for (int i = 0; i < s.length(); i++)
                  System.out.println(s.charAt(" + i + ") = " +
                                          s.charAt(i) + "\t\ta[" + i +
                                          "] = " + a[i]);
      }
}
```

La salida sería:

```
s = ABCDEFG                a = ABCDEFG
s.length() = 7             a.length = 7
s.charAt(0) = A            a[0] = A
s.charAt(1) = B            a[1] = B
s.charAt(2) = C            a[2] = C
s.charAt(3) = D            a[3] = D
s.charAt(4) = E            a[4] = E
s.charAt(5) = F            a[5] = F
s.charAt(6) = G            a[6] = G
```

Los arreglos tienen un campo público llamado `length`, el cual almacena la cantidad de elementos en el arreglo. Al igual que los objetos `String`, los arreglos utilizan la indicación basada en cero. Esto significa que el primer elemento tiene índice 0.

Ejemplo 8.2: Un método para eliminar todas las ocurrencias de un caracter en una cadena

```
public class PruebaEliminarCaracteres {
      public static String eliminar(String s, char c) {
            int n = s.length();
            char[] a = new char[n];
            int i = 0, j = 0;

            while (i+j < n) {
                  char sc = s.charAt(i+j);
                  if (sc == c)
                        i++;
                  else
                        a[i++] = sc;
            }

            return new String(a,0,i);
      }

      public static void main(String[] args) {
            String s = new String("ABRACADABRA");
            System.out.println(s);
            s = eliminar(s,'A');
            System.out.println(s);
      }
}
```

La salida sería:

```
ABRACADABRA
BRCDBR
```

8.3. Propiedades de los arreglos en Java

En Java se pueden crear arreglos cuyos elementos sean de alguno de los ocho tipos primitivos o referencia. La sintaxis es la siguiente:

```
Tipo-elemento[] nombre-arreglo;
nombre-arreglo = new Tipo-elemento[Tamaño];
```

Al igual que con los objetos, la declaración y creación del arreglo puede ser combinada en un solo enunciado.

```
Tipo-elemento[] nombre-arreglo = new Tipo-elemento[Tamaño];
```

Algunos ejemplos son:

```
float[] x;
x = new float[8];

boolean[] banderas = new boolean[1024];
```

```
String[] nombres = new String[32];
Punto[] linea = new Punto[12];
```

Cuando el tipo de elemento es una referencia (como en el caso de objetos String), el alojamiento se hace sólo de referencias a objetos de esa clase. No se alojan los objetos en si.

```
int[] enteros = {44, 88, 55, 33};
```

Un arreglo puede ser inicializado usando una lista de inicialización como se muestra en el ejemplo anterior.

8.4. Copia de arreglos

En Java no se pueden copiar arreglos utilizando el operador de asignación, esto causaria que se copie la referencia al arreglo más no los elementos. Para esto Java provee un método universal especial. Este método pertenece a la clase `System` y su firma es la siguiente:

```
public static void arraycopy(Object fuente, int posFuente,
                                    Object destino, posDestino,
                                    int cantidad)
```

Este método copia `cantidad` elementos del arreglo `fuente` al arreglo `destino`. `posFuente` y `posDestino` indican las posiciones iniciales para realizar la copia en los arreglos `fuente` y `destino`, respectivamente.

8.5. La clase `Vector`

El tipo de elemento de un arreglo puede ser cualquiera de los ocho tipos primitivos de Java o una referencia. En el caso de un arreglo de un tipo referencia, por las propiedades de herencia, cada elemento puede ser una referencia a un objeto de cualquier sublcase de la clase.

Por ejemplo:

```
lista = new Persona[4];
lista[0] = new Persona( ... );
lista[1] = new Estudiante( ... );
lista[2] = new Empleado( ... );
lista[3] = new Abogado( ... );
```

A pesar de que cada uno de los elementos del arreglo es, a traves de la herencia, un objeto `Persona`, obtenemos una lista heterogenea. Este es un ejemplo de la característica de la programación orientada a objetos llamada *polimorfismo*.

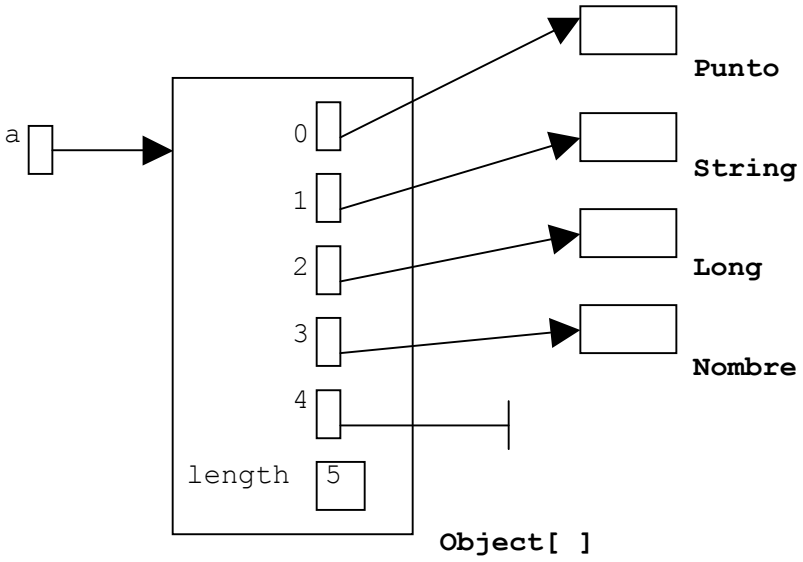
Ejemplo 8.3: Un arreglo de objetos `Object`

```
class PruebaArregloDeObjet {
      public static void main(String[] args) {
            Object[] a = new Object[5];
            a[0] = new Punto(2,3);
            a[1] = new String("Hola mundo");
            a[2] = new Long(44);
            a[3] = new Nombre("Guillermo", "Licea", "Sandoval");

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

La salida sería:

```
a[0] = (2.0, 3.0)
a[1] = Hola Mundo
a[2] = 44
a[3] = Guillermo Licea Sandoval
a[4] = null
```

Cada elemento del arreglo `a` es una referencia a un objeto de tipo `Object`, pero a través de la herencia, cualquier objeto en Java es un `Object`, de tal manera que el arreglo a puede almacenar cualquier cosa.



Como ya se ha visto, cada una de las ideas interesantes en Java es encapsulada en una clase. La idea de tener un arreglo universal se encapsuló en la clase `Vector`. Un objeto de tipo `Vector` es un arreglo universal que incrementa su tamaño automáticamente cuando lo necesita.

La clase `Vector` está incluida en el paquete `java.util`, así que es necesario incluir el enunciado `import java.util.Vector` para poder utilizarla.

Ejemplo 8.4: La lista de números telefónicos utilizando un `Vector`

```
import java.util.Vector;

class PruebaAmigos {
      public static void main(String[] args) {
            Vector amigos = new Vector();
            amigos.addElement(new Amigo("Silvio Rodriguez","123-4567");
            amigos.addElement(new Amigo("Pablo Milanes","890-1234");
            amigos.addElement(new Amigo("Pedro Guerra","567-8901");
            System.out.println(amigos);
      }
}
```

La salida sería:

```
[Silvio Rodriguez: 123-4567, Pablo Milanes: 890-1234, Pedro Guerra: 567-8901]
```

Un objeto `Vector` en Java es realmente una lista dinámica. Los elementos pueden ser agregados o removidos de cualquier posición en la lista. A continuación se presentan algunos de los métodos de la clase `Vector`.

```
void addElement(Object o)      // agrega un objeto a la lista
boolean contains(Object o)     // regresa true si el objeto se encuentra
                               // en la lista
Object elementAt(int i)        // regresa el objeto en la posicion i de
                               // la lista
Object firstElement()          // regresa una referencia al primer objeto
                               // en la lista
int indexOf(Object o)          // regresa el indice (o -1) de la primera
                               // ocurrencia del objeto
int indexOf(Object o, int i)   // regresa el indice (o -1) de la primera
                               // ocurrencia del objeto, iniciando la
                               // busqueda en la posicion indicada
void insertElementAt(Object o, int i)      // agrega un objeto en la
                                           //posicion indicada
boolean isEmpty()              // regresa true si la lista esta vacia
Object lastElement()           // regresa una referencia al ultimo objeto
                               // en la lista
int lastIndexOf(Object o)      // regresa el indice de la ultima
                               // ocurrencia del objeto
int lastIndexOf(Object o, int i)     // regresa el indice de la ultima
                                     // ocurrencia del objeto, indicando
                                     // la busqueda hacia atrás en la
                                     // posicion indicada
void removeAllElements()       // elimina todos los objetos de la lista
boolean removeElement(Object o)      // elimina la primera ocurrencia del
                                     // objeto en la lista, regresa true
                                     // si encontro el objeto
void removeElement(int i)      // elimina el objeto en la posicion
                               // indicada
int size()                     // regresa el numero de elementos que
                               // contiene la lista
```

Ejemplo 8.5: Reacomodando los elementos de un `Vector`

```
import java.util.Vector;

class PruebaAmigos {
      public static void main(String[] args) {
            Vector amigos = new Vector();

            amigos.addElement(new Amigo("Silvio Rodriguez","123-4567");
            amigos.addElement(new Amigo("Pablo Milanes","890-1234");
            amigos.addElement(new Amigo("Pedro Guerra","567-8901");
            System.out.println(amigos);
            amigos.insertElementAt(amigos.elementAt(2),0);
            System.out.println(amigos);
            amigos.removeElementAt(3);
            System.out.println(amigos);
      }
}
```

La salida sería:

```
[Silvio Rodriguez: 123-4567, Pablo Milanes: 890-1234, Pedro Guerra: 567-8901]
[Pedro Guerra: 567-8901, Silvio Rodriguez: 123-4567, Pablo Milanes: 890-1234,
 Pedro Guerra: 567-8901]
[Pedro Guerra: 567-8901, Silvio Rodriguez: 123-4567, Pablo Milanes: 890-1234]
```

A diferencia de un arreglo que tiene una longitud constante, un `Vector` tiene tamaño, el cual es el número de referencias a objetos que contiene. Este número es dinámico, cambia cada vez que es agregado o eliminado un objeto.

Además de su tamaño, un `Vector` tiene capacidad, la cual es el número de espacios alojados para almacenar referencias a objetos. Este número es siempre igual o mayor que su tamaño.

Ejemplo 8.6: Probando el tamaño y capacidad de un `Vector`

```
import java.util.Vector;

class PruebaTamanoYCapacidad {
      public static void main(String[] args) {
            Vector v = new Vector();

            for (int i=0; i<=30; i++) {
                  v.addElement(new Integer(i));
                  imprimir(v);
            }
      }

      static void imprimir(Vector v) {
            System.out.println("v = " + v);
            System.out.println("v.size() = " + v.size());
            System.out.println("v.capacity() = " + v.capacity());
      }
}
```

Cuando la capacidad de un `Vector` cambia, el `Vector` debe reconstruirse, es decir, es necesario alojar un nuevo bloque de memoria para mover el `Vector`. Para evitar que el `Vector` se reconstruya varias veces, es posible indicar explícitamente la capacidad. Esto puede hacerse al momento de crear el `Vector` o a través del método `ensureCapacity()`.

Ejemplo 8.7: Cambiando la capacidad de un `Vector`

```
import java.util.Vector;

class PruebaCapacidad {
      public static void main(String[] args) {
            Vector v = new Vector();

            v.ensureCapacity(50);
            for (int i=0; i<=30; i++) {
                  v.addElement(new Integer(i));
                  imprimir(v);
            }
      }

      static void imprimir(Vector v) {
            System.out.println("v = " + v);
            System.out.println("v.capacity() = " + v.capacity());
      }
}
```

8.6. Arreglos bidimensionales

Un arreglo bidimensional es aquel que usa dos subíndices en vez de uno. Podemos imaginar el arreglo como una rejilla de filas y columnas, donde el primer subíndice se refiere a las filas y el segundo a las columnas.

Un arreglo bidimensional es en realidad un arreglo de arreglos.

Ejemplo 8.8: Un arreglo de arreglos

```
class PruebaArreglo {
      public static void main(String[] args) {
            int[][] a = new int[7][9];
            System.out.println("a.length = " + a.length);
            System.out.println("a[0].length = " + a[0].length);
      }
}
```

La salida sería:

```
a.length = 7
a[0].length = 9
```

Un arreglo bidimensional puede ser inicializado de la misma manera que un arreglo unidimensional. La diferencia es que la lista de inicialización será una lista de listas.

Ejemplo 8.9: Inicializando un arreglo de arreglos

```
class PruebaInicializacionDeArreglo {
      public static void main(String[] args) {
            int[][] a = { {77, 33, 88}, {11, 55, 22, 99}, {66, 44} };

            for (int i=0; i<a.length; i++) {
                  for (int j=0; j<a[i].length; j++)
                        System.out.print("\t" + a[i][j]);

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

La salida sería:

```
77    33    88
11    55    22    99
66    44
```

Usando la misma idea, podemos trabajar con arreglos tridimensionales, tetradimensionales o n-dimensionales.

Preguntas de repaso

- ¿ Porque es diferente el tamaño de un arreglo y de un `String` ?
- ¿ Cual es la diferencia entre accesar elementos de un arreglo de caracteres y elementos de un *`String`* ?
- ¿ Por qué los arreglos son procesados a través de enunciados `for` ?
- ¿ Porque un arreglo de `Object` es llamado un arreglo universal ?
- ¿ Cuál es la diferencia entre el tamaño y la capacidad de un `Vector` ?
- ¿ Cual es la diferencia entre un `Vector` y un arreglo de objetos ?
- ¿ Cuál es la diferencia entre un `String` y un arreglo de caracteres ?

Problemas de programación

- Implemente los siguientes métodos:

```
static double suma(double[] x)                  // regresa la suma de los elementos del arreglo
static double max(double[] x)                   // regresa el mayor de los elementos del arreglo
static double rango(double[] x)                 // regresa la diferencia entre el menor y el mayor de los
                                                // elementos del arreglo
static boolean sonInguales(double[] x, double[] y)      // regresa true si los dos arrreglos son iguales
static boolean sonInguales(double[][] x, double[][] y)  // regresa true si los dos arreglos son iguales
static void intercambiar(Vector v, int I, int j)  // intercambia los elementos del Vector en las posiciones i, j
int cuentaDistintos()                           // regresa el numero de objetos referenciados en la lista
static void traspuesta(double[][] x)            // regresa la matriz traspuesta de x
static int[] fibonacci(int n)                   // regresa los primeros n+1 numeros de fibonacci
static int[] primos(int n)                      // regresa los primeros n+1 numeros primos
```