EL LENGUAJE DE PROGRAMACION

JAVA

AUTOR:

ING. RAUL CORDOVA, M.Sc.

FEBRERO − 2000

I. FUNDAMENTOS DE JAVA

1. Enunciados y expresiones

- Un enunciado es lo más sencillo que hay en Java; cada uno forma una sola operación Java.
- Ejemplos:

```
int i = 1;

import java.awt.Font;

System.out.println(Esta moto es una  + color +   + fabricante);

m.engineState 0 true;
```

- Algunos enunciados regresan valores. Este tipo de enunciados se denominan expresiones.
- Cada enunciado termina en punto y coma.
- Existen enunciados compuestos o bloques, los cuales pueden colocarse en cualquier lugar donde pondría un solo enunciado.
- Los enunciados de bloque están rodeados por llaves.

2. Variables y tipos de datos

- Variables son lugares en la memoria donde pueden guardarse valores.
- Poseen un nombre, un tipo y un valor.
- Antes de usar una variable, primero se la debe declarar y a partir de ahí es factible asignarle valores.
- Java posee tres clases de variables: de instancia, de clase y locales.
- Las variables de instancia se utilizan para definir atributos o el estado de un objeto en particular.
- Las variables de clase son similares a las de instancia, con la diferencia de que sus valores se aplican a todas las instancias de clase (y a la misma clase), en lugar de tener diferentes valores para cada objeto.
- Las variables locales se declaran y utilizan dentro de las definiciones de método, por ejemplo, para contadores de índica en ciclos, como variables temporales, o para guardar valores que sólo se necesitan dentro de la definición.
- También pueden usarse dentro de bloques. Una vez que el método o bloque termina su ejecución, la definición de variable y su valor dejan de existir.
- Se recomienda utilizar variables locales para guardar la información necesitada por un solo método y las variables de instancia para guardar la información requerida por varios métodos en el objeto.

Declaración de variables

- Consisten en un tipo y nombre de variable. Ejemplo:

```
int myAge;

String myName;

boolean isTired;
```

- Las definiciones de variables pueden ubicarse en cualquier lugar de la definición del método (es decir, en cualquier lugar donde pueda ubicarse un enunciado normal de Java), aunque por lo común, se declaran al inicio de la definición, antes de utilizarse:

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

int count;

String title;

boolean isAsleep;

...

}
```

Puede encadenar nombres de variables con el mismo tipo:

```
int x,y,z;

String firstName, LastName;
```

También puede dar a cada variable un valor inicial cuando la declare:

```
int myAge, mySize, numShoes = 28;

String myName = Gina;

boolean isTired = true;

int a = 4, b = 5, c = 6;
```

- Si existen variables múltiples en la misma línea y sólo una está inicializada, el valor inicial se aplica sólo a la última variable en una declaración.
- También se pueden agrupar en la misma línea variables individuales e inicializadas si se utilizan comas, como en el último ejemplo.
- Se deben dar valores a las variables locales antes de utilizarlas, sino el programa en Java no se compilará). Por esta razón, siempre es recomendable dar valores iniciales a las variables locales.
- Las definiciones de variables de instancia y de clase no tienen esta restricción: su valor inicial depende del tipo de variable: null para instancias de clase, 0 para variables numéricas, `\' para caracteres y false para booleanas.

Nombres de variables

- Los nombres de variables pueden iniciar con una letra, un guión de subrayado (_) o un signo de dólares ($).
- No pueden comenzar con un número.
- Después del primer carácter, los nombres de variables pueden incluir cualquier letra o número.
- En Java los símbolos como %, *, @ entre otros, con frecuencia están reservados para los operadores, así que se debe tener cuidado al momento de emplear símbolos en nombres de variables.
- Java también usa el conjunto de caracteres Unicode. Esta es una definición del conjunto de signos que no sólo ofrece caracteres en el conjunto estándar ASCII, sino también varios millones de caracteres para representar la mayoría de los alfabetos internacionales.
- Esto significa que se pueden usar caracteres acentuados y otros glifos, así como caracteres legales en nombres de variables, siempre que cuenten con un número de carácter Unicode sobre 00C0.
- Java es sensible al tamaño de las letras, lo cual significa que las mayúsculas son diferentes de las minúsculas. En otras palabras, la variable X es diferente a la variable x, y que rosa no es Rosa, ni ROSA.
- Por convención, las variables Java tienen nombre significativos, con frecuencia formados de varias palabras combinadas.
- La primera letra está en minúsculas, pero las siguientes tienen su letra inicial en mayúsculas:

Button theButton;

long reallyBigNumber;

boolean currentWeatherStateOfPlanetXShortVersion;

Tipos de variables

- Además de los nombres de variables, cada declaración de variable debe tener un tipo, el cual define los valores que esa variable puede mantener.
- El tipo de variable puede ser uno de estos tres:
- Uno de los ocho tipos de datos primitivos.
- El nombre de una clase o interfaz.
- Un arreglo.

- Los ocho tipos de datos primitivos manejan tipos comunes para enteros, números de punto flotante, caracteres, y valores booleanos (cierto o falso).
- Se llaman primitivos porque están integrados en el sistema y no son objetos en realidad, lo cual hace su uso más eficiente.
- Estos tipos de datos son independientes de la computadora, por lo que se puede confiar en que su tamaño y características son consistentes en los programas Java.
- Existen cuatro tipos de enteros Java, cada uno con un rango diferente de valores, como se muestra en la Tabla 1.
- Todos tienen signo, por lo cual pueden mantener números positivos o negativos.
- El tipo que se seleccione para las variables dependerá del rango de valores que se espera mantenga esa variable.
- Si un valor es demasiado grande para un tipo de variable, se truncará sin saberlo.

TABLA 1. TIPOS DE ENTEROS

Tipo Tamaño Rango

byte 8 bits −128 a +127

short 16 bits −32.768 a +32.767

int 32 bits −2.147.483.648 a +2.147.483.647

long 64 bits −9.223.372.036.854.775.808 a +9.223.372.036.854.775.808

- Los números de punto flotante se utilizan para valores con una parte decimal. Estos cumplen con el estándar IEEE754 (un estándar internacional para definir números de punto flotante y su aritmética).
- Existen dos tipos de punto flotante: float (32 bits, precisión sencilla) y double (64 bits, doble precisión).
- El tipo **char** se utiliza para caracteres individuales. Puesto que Java emplea el conjunto de caracteres Unicode, el tipo char tiene 16 bits de precisión, sin signo.
- El tipo **boolean** puede tener uno de dos valores: true o false. Advertirá que a diferencia de otros lenguajes similares a C, boolean no es un número, ni tampoco puede tratarse como si lo fuera.
- Además de los 8 tipos de datos básicos, las variables en Java también pueden declararse para mantener una instancia de una clase en particular:

String apellido;

Font fuenteBásica;

OvalShape myOval;

- Cada una de estas variables sólo puede mantener instancias de una clase dada. Al crear nuevas clases también se pueden declarar variables para mantener instancias de esas clases (y sus subclases).
- En Java, para declarar nuevos tipos de datos, primero se debe declarar una clase nueva y después las variables pueden declararse para que sean de ese tipo de clase.

ASIGNACION DE VALORES A VARIABLES

- Se usa el operador de asignación = :

tamaño = 14;

matriculado = true;

Comentarios

- En Java se tienen tres tipos:

/* */ para varias líneas

// para una sola línea

/** */ usada en javadoc para generar documentación API del código.

Literales

Se usan para indicar valores en los programas.

Literales de número

- Existen varias literales de enteros:

Entera decimal de tipo int . Ej. 4 (puede ser también byte o short)

Entera decimal más grande que int es en forma automática long.

Un número pequeño a long, se agrega una L o l al final del número: 4L

Enteros negativos están precedidos por un signo menos: −45

Enteros octales: empiezan con cero. Ej. 0777 ó 0004.

Enteros hexadecimales: empiezan con 0x ó 0X. Ej. 0xFF, 0XAF45. Pueden contener dígitos normales (0−9) o hexadecimales en mayúsculas o minúsculas (a−f o A−F).

Literales de punto flotante tienen dos partes: entera y decimal. Ej. 5.6777777.

Las literales de punto flotante resultan en un número de punto flotante de tipo double sin importar la precisión de ese número. Se puede forzar el número al tipo float al agregar la letra f (o F) al número. Ej: 2.56F.

Se pueden también usar exponentes en las literales de punto flotante usándose e o E, seguida del exponente, positivo o negativo. Ej. 10e45, .36E−2.

Literales booleanas

- Consisten en las palabras clave true o false, las cuales se pueden usar en cualquier lugar en que se necesite hacer una prueba o como los únicos valores posibles para variables booleanas.

Literales de número

- Se expresan por un solo carácter entre comillas sencillas: `a', `#', `3'.
- Se guardan como caracteres Unicode de 16 bits.
- En Java también se tienen códigos especiales, no imprimibles o Unicode.

TABLA: Códigos de escape de caracteres.

Escape Significado

\n Línea nueva

\t Tabulador

\b Retroceso

\r Regreso de carro

\f Alimentación de forma

\\ Diagonal inversa

\' Comilla sencilla

\ Comillas dobles

\ddd Octal

\xdd Hexadecimal

\udddd Carácter Unicode

Literales de cadena

- Una combinación de caracteres es una cadena. Las cadenas en Java son instancias de la clase String.
- Las cadenas no son simples arreglos de caracteres como lo son en C o C++, aunque cuentan con características parecidas a las de los arreglos, como verificar su longitud y cambiar caracteres individuales.
- Puesto que los objetos de cadena en Java son objetos reales, cuentan con métodos que permiten combinar, verificar y modificar cadenas con gran facilidad.
- Las literales de cadena consisten en una serie de caracteres dentro de comillas dobles:

Hola, yo soy un literal de tipo string.

// una cadena vacía

- Las cadenas pueden tener constantes de caracteres como línea nueva, tabulación y caracteres Unicode:

Una cadena con un carácter de tabulador \t en ella

Cadenas anidadas son \cadenas dentro de\ otras cadenas

Esta cadena produce el carácter de marca registrada con código Unicode \u2122

Expresiones y operadores

- Las expresiones son la forma más sencilla de un enunciado en Java.
- Las expresiones son enunciados que regresan un valor.
- Los operadores son símbolos especiales que, por lo común, se utilizan en expresiones.
- Los operadores en Java incluyen a los aritméticos, varias formas de asignación, incrementos y decrementos, así como operaciones lógicas.

Operadores aritméticos

Operador Significado Ejemplo

+ Suma 3 + 4

− Resta 5 − 7

* Multiplicación 5 * 5

/ División 14/7

% Módulo 20 % 7

- Cada operador toma dos operandos, uno a cada lado del operador. El operador de resta (−) puede usarse para negar un operando sencillo.
- La división de enteros resulta en un entero y el residuo se ignora. Ej: 31/9 da 3.
- El módulo proporciona el residuo de una división entera. 31%9 da 4.
- Un entero operado con otro entero da un entero.
- Si un operando es long, el resultado será long.
- Si un operando es entero y otro de punto flotante, el resultado será de punto flotante.

Programa de ejemplo:

```
class PruebaAritmética {

public static void main (String args[ ]) {

short x = 6;

int y = 4;

float a = 12.5f;

float b = 7f;

System.out.println(x es  + x + , y es  + y);

System.out.println(x − y =  + (x − y));

System.out.println(x * y =  + (x * y));

System.out.println(x / y =  + (x / y));

System.out.println(x % y =  + (x % y));

System.out.println(a es  + a + , b es  + b;

System.out.println(a / b =  + (a / b));

}

}
```

Resultado:

```
x es 6, y es 4

x + y = 10

x − y = 2

x * y = 24

x / y = 1
```

x % y = 2

a es 12.5, b es 7

a / b = 1.78571

HERENCIA, INTERFACES Y PAQUETES

- Son mecanismos para organizar clases y el comportamiento de la clase.

HERENCIA

- Con la herencia, todas las clases, las que se programan, las de otras bibliotecas que se utilicen y también las clases de la utilería estándar, están arregladas dentro de una jerarquía estricta.
- Cada clase tiene una superclase (la clase superior en la jerarquía) y puede tener una o más subclases (las clases debajo de esa clase en la jerarquía).
- Las clases inferiores en la jerarquía heredan de las clases más altas.
- Las subclases heredan todos los métodos y variables de las superclases, con lo que no se requiere volver a definir o copiar ese código de alguna otra clase.
- En la parte superior de la jerarquía de clase Java está la clase *Object*.
- Todas las clases heredan de esta superclase.
- Es la clase más general en la jerarquía, ya que define el comportamiento heredado por todas las clases en la jerarquía de clase Java.
- Cada clase hacia abajo en la jerarquía agrega más información y se vuelve más apta para un propósito específico.
- Cuando se escriben nuevas clases Java, por lo general se deseará crear una clase que tenga toda la información que otra clase tiene, además de alguna información extra.
- Por ejemplo, tal vez se desee una versión de un *Button* con su propia etiqueta integrada. Para obtener toda la información de esta clase, lo que debe hacer es definir la clase que hereda de la clase Button.
- La nueva clase, en forma automática obtendrá el comportamiento definido en *Button* (y en las superclases de ésta), así que su única preocupación deben ser los elementos que diferencian a su clase de la clase Button original.
- Este mecanismo utilizado para definir nuevas clases así como las diferencias entre ellas y sus supeclases, se llama subclasificación.
- La subclasificación implica crear una nueva clase que heredará de alguna otra clase en la jerarquía. Al usar la subclasificación, sólo se necesitarán definir las diferencias entre esta clase y su padre; el comportamiento adicional está disponible para la nueva clase mediante la herencia.
- Si una clase define un comportamiento nuevo por completo y no es en sí una subclase de otra clase, también puede heredar directamente de Object, y todavía se permite ajustarse con habilida a la jerarquía de clase Java.
- De hecho, si se crea una definición de clase que no indica su superclase en la primera línea, Java de manera automática supone que está heredando de Object.
- La clase Motorcycle que se creó anteriormente, heredaba de esta clase.

CREACION DE UNA JERARQUÍA DE CLASES

- Si se va crear un conjunto más grande de clases, tiene sentido que estas clases no sólo hereden de la jerarquía de clases existente, sino que también hagan por sí mismas una jerarquía.
- Esto puede implicar una planeación previa al intentar organizar el código Java, aunque las ventajas son significativas una vez que ésta se realiza:

- Cuando se desarrollan clases en una jerarquía, se puede descomponer información común para

múltiples clases en superclases y después volver a utilizar la información de esa superclase una y otra vez. Cada subclase obtendrá esa información común de su superclase.
- Cambiar o insertar una clase hacia arriba en la jerarquía, de manera automática modifica el comportamiento de las clases inferiores; no hay necesidad de cambiar o volver a compilar alguna de las clases más bajas, ya que se obtiene la nueva información por medio de la herencia y no al copiar algo del código.

INTERFACES Y PAQUETES

- Java posee herencia sencilla, no múltiple como C++
- INTERFAZ es una colección de nombres de métodos sin definiciones reales que indican que una clase tiene un conjunto de comportamientos, además de los que la clase hereda de sus superclases.
- Aunque una clase Java puede tener sólo una superclase (a causa de la herencia sencilla), esa clase también puede implantar cualquier número de interfaces. Al implantar una interfaz, una clase proporciona implantaciones de métodos (definiciones) para los nombres de métodos definidos por la interfaz.
- Si dos clases muy diferentes implantan la misma interfaz, ambas pueden responder a la misma llamada del método (definido por la interfaz respectiva), aunque lo que cada clase hace en respuesta a esas llamadas del método puede ser muy distinto.
- PAQUETES en Java son una manera de agrupar clases e interfaces relacionadas. Permiten que grupos modulares de clases estén disponibles sólo cuando se necesitan, y eliminan conflictos potenciales entre los nombres de clase, en diferentes grupos de clase.

Otros puntos importantes son:

- Las bibliotecas de clase en el Java Developer´s Kit (JDK), están contenidas en un paquete llamado java. Las clases del paquete están garantizadas para encontrarse disponibles en cualquier implantación Java, y son las únicas clases que garantizan su disponibilidad para diferentes implantaciones. El paquete java contiene otros paquetes para clases que definen el lenguaje de las clases de entrada y salida, algunos fundamentos acerca de redes y las funciones del kit de herramientas para ventanas. Las clases contenidas en otros paquetes (por ejemplo, las clases en los paquetes **sun** o **netscape**), pueden estar disponibles sólo en implantaciones específicas.
- De manera predeterminada, las clases Java tienen acceso sólo a las clases que se encuentran en java.lang (el paquete de lenguaje básico dentro del paquete java). Para utilizar clases de cualquier otro paquete, se tiene que referir a ellas en forma explícita por el nombre del paquete o importarlas al archivo fuente.
- Para referirse a una clase dentro del paquete, se deben listar todos los paquetes donde está contenida esa clase y el nombre de la clase, todo separado por puntos(.). Por ejemplo, tome la clase Color, la cual está contenida en el paquete awt (Abstract Windowing Toolkit − Kit de herramientas para ventanas abstractas), el cual, a su vez, se encuentra dentro del paquete java. Por lo tanto, para referirse a la clase Color en su programa, utilice la notación java.awt.Color.

CREACION DE SUBCLASES

- A continuación, se va a crear una clase que sea una subclase de otra clase y a sobreponer algunos métodos. También se obtendrá conocimientos básicos del funcionamiento de los paquetes en este ejemplo.
- En el ejemplo, se creará un applet.

```
public class HelloAgainApplet extends java.applet.Applet {

}
```

- Aquí se creó una clase llamada HelloAgainApplet.
- La parte que dice **extends java.applet.Applet**, indica que la clase de applet creada es una subclase de la clase Applet.
- Ya que la clase Applet está dentro del paquete java.applet, no se tendrá acceso automático a esa clase y se tendrá que referirse a ella en forma explícita por paquete y nombre de clase.
- La otra parte de esta definición de clase es la palabra clave **public**. Significa que la clase creada está disponible para todo el sistema Java después de que se cargó.
- En la mayoría de ocasiones, se necesitará hacer una clase public sólo si se desea que sea visible para todas las demás clases del programa Java, aunque los applets, en particular, deben declararse como públicos.
- Incorporemos una variable de instancia para contener a un objeto Font:

```
Font f = new Font(TimesRoman, Font.BOLD, 36);
```

- La variable de instancia f ahora contiene una nueva instancia de la clase Font que forma parte del paquete java.awt. Este objeto fuente en particular es una fuente Times Roman, en negritas y de 36 puntos de altura.
- Al crear una variable de instancia para guardar este objeto fuente, lo vuelve disponible para todos los métodos en su clase. Ahora vamos a crear un método que lo utilice.
- Cuando escribe applets, hay varios métodos estándares definidos en las superclases del applet que por lo común sobrepondrá en la clase de su applet. Estos incluyen métodos para inicializar el applet, para empezar su ejecución, para manejar operaciones como movimientos o clics del ratón, o para limpiar el espacio cuando el applet detenga su ejecución.
- Uno de estos métodos estándares es paint(), el cual de hecho despliega su applet en la pantalla. La definición predeterminada de paint() no efectúa nada: es un método vacío. Al sobreponer este método, le indica al applet qué debe dibujar en la pantalla.
- Veamos una definición de paint():

```
public void paint(Graphics g) {

g.setFont(f);

g.setColor(Color.red);

g.drawString(Hello again!, 5, 25);

}
```

- Respecto al método paint(), obsérvese que está declarado como public, al igual que el applet mismo. Este método es así por un motivo diferente: a causa de que el método al que sobrepone está declarado de la misma forma. Si un método de una superclase está definido como público, el método que lo sobrepone también debe ser público o, de lo contrario, tendrá un error cuando compile la clase.
- En segundo lugar, el método paint() toma un solo argumento: una instancia de la clase Graphics. Esta clase tiene un comportamiento independiente de la plataforma para presentar fuentes, colores y operaciones de dibujo básicas.
- Dentro del método paint(), se realizaron tres actividades:
- Se le indicó al objeto gráficos que la fuente de dibujo predeterminada sería la contenida en la variable de instancia f.
- Se le indicó al objeto gráficos que el color prederminado es una instancia de la clase Color para el color rojo.
- Por último, se dibujó la cadena Hello Again! en la pantalla, en las posiciones x y de 5 y 25.
- Para un applet tan sencillo, esto es todo lo que necesita hacer. Aquí está la apariencia del applet hasta

ahora:

```
public class Hello AgainApplet extends java.applet.Applet {

Font f = new Font(TimesRoman, Font.BOLD,26);

public void paint (Graphics g) {

g.setFont(f);

g.setColor(Color.red);

g.drawString(Hello again!, 5, 50);

}

}
```

- Para poder usar Graphics, Font y Color, es necesario importarlas al inicio del archivo de clase, antes de la defínción de clase.

```
import java.awt.Graphics;

import java.awt.Font;

import java.awt.Color;
```

- También se puede importar un paquete completo de clases (public), al utilizar un asterisco (*) en lugar del nombre de la clase específica. Por ejemplo, para importar todas clases en el paquete awt, se puede usar esta línea:

```
import java.awt.*;
```

- La clase completa lucirá entonces así:

```
import java.awt.Graphics;

import java.awt.Font;

import java.awt.Color;

public class Hello AgainApplet extends java.applet.Applet {

Font f = new Font(TimesRoman, Font.BOLD,26);

public void paint (Graphics g) {

g.setFont(f);

g.setColor(Color.red);

g.drawString(Hello again!, 5, 50);
```

```
}

}
```

- Ahora, con las clases adecuadas importadas a su programa, HelloAgainApplet deberá compilarse con limpieza a un archivo de clase.
- Para probarlo, creemos un archivo HTML con la etiqueta <APPLET>:

```
<HTML>

<HEAD>

<TITLE>Applet<TITLE>

</HEAD>

<BODY>

<P>El applet Java dice:

<APPLET CODE=HelloAgainApplet.class WIDTH=200 HEIGHT=50>

</APPLET>

</BODY>

</HTML>
```

- Para este ejemplo HTML, el archivo de clase Java deberá estar en el mismo directorio de este archivo HTML.
- Guardar el archivo en HelloAgainApplet.html y ejecutar su visualizador con capacidad Java o el appletviewer Java.

CAPITULO 4: TRABAJO CON OBJETOS

Creación de objetos

Para crear un nuevo objeto, se utiliza el operador **new**, con el nombre de la clase de la cual se desea crear una instancia, seguido de paréntesis:

```
String str = new String();

Random r = new Random();

Motorcycle m2 = new Motorcycle();
```

Ejemplo: creación de objetos de la clase Date con el uso de new:

```
import java.util.Date;

class CrearFechas {
```

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

Date d1,d2, d3;

d1 = new Date();

System.out.println(Fecha 1:  + d1);

d2 = new Date(71,7,1,7,30);

System.out.println(Fecha 2:  + d2);

d3 = new Date(April 3 1993 3:24 PM);

System.out.println(Fecha 3:  + d3);

}

}
```

SALIDA:

```
Fecha 1: Tue Feb 13 09:36:56 PST 1996

Fecha 2: Sun Aug 01 07:30:00 PDT 1971

Fecha 3: Sat Apr 03 15:24:00 PST 1993
```

- Aquí se dan tres maneras diferentes de instanciar a la clase Date, utilizando diferentes argumentos para new.
- Esto es posible gracias a que se han definido tres métodos diferentes para instanciar a la clase Date, cada uno de los cuales se llama constructor.
- El uso de new hace que, primero, se cree la nueva instancia de la clase dada y se le asigna memoria.
- Luego, cuando se crea un nuevo objeto, se llama a un método especial definido en la clase dada, llamado constructor.
- Los constructores son métodos especiales para crear e inicializar nuevas instancias de clase.
- Inicializan el nuevo objeto y sus variables, crean cualquier otro objeto que se necesite y pueden realizar cualquier otra operación para inicializarse a sí mismos.
- Cada definición de constructor puede tener un número diferente o tipos de argumentos diferentes; así, cuando se utilice new se puede especificar diferentes argumentos en la lista de argumentos y se llamará al constructor adecuado para esos argumentos.

ADMINISTRACION DE MEMORIA EN JAVA

- Es dinámica y automática. Cuando se crea un nuevo objeto, éste, en forma automática destina la cantidad adecuada de memoria para ese objeto en la pila.
- El programador no tiene que asignar memoria para algún objeto en forma explícita, pues el lenguaje lo hace por sí mismo.
- Cuando se deja de usar el objeto, ya no contará con referencias a ese objeto: la administración de la memoria es automática.
- Java, para hacer esto, cuenta con un colector de basura que localiza objetos no usados y libera la memoria que emplearon esos objetos.

- No es necesario hacer una liberación explícita de memoria.

ACCESO CONFIGURACIÓN DE VARIABLES DE INSTANCIA Y DE CLASE

- Las variables de instancia se comportan de la misma forma que las locales, pero para referirse a ellas, se usará una notación punto.

Obtención de valores de variables de instancia

- Se usa la notación punto, en la cual un nombre de variable de instancia o de clase tendrá dos partes: el objeto, al lado izquierdo del punto y la variable, al lado derecho.
- Ej: myObject.var;
- Esta forma de accesar a las variables es una expresión (regresa un valor).
- Se puede anidar el acceso a las variables de instancia. Si var mantiene un objeto y éste tiene su propia variable de instancia llamada state, se puede referir a ella de esta manera:

myObject.var.state;

- Las expresiones de punto se evalúan de izquierda a derecha, así que se debe empezar con la variable var de myObject, la cual se dirige a otro objeto con la variable state. Al final se tendrá el valor de esa variable.

Cambio de valores

- Sólo se inserta un operador de asignación al lado derecho de la expresión:

myObject.var.state = true;

Ejemplo: este programa prueba y modifica las variables de instancia de un objeto Point, el cual es parte del paquete java.awt y se refiere a un punto de coordenadas con valores x Y y.

```
import java.awt.Point;

class TestPoint {

public static void main (String args[ ]) {

Point elPunto = new Point(10,10);

System.out.println(X es  + elPunto.x);

System.out.println(Y es  + elPunto.y);

System.out.println(Seteando X a 5.);

elPunto.x = 5;

System.out.println(Seteando Y a 15.);

elPunto.y = 15;

System.out.println(X es  + elPunto.x);
```

```
System.out.println(Y es  + elPunto.y);

}

}
```

SALIDA:

X es 10

Y es 10

Seteando X a 5.

Seteando Y a 15.

X es 5

Y es 15

VARIABLES DE CLASE

- Estas variables se definen y almacenan en la misma clase. Sus valores se aplican a la clase y a todas sus instancias.
- Con estas variables, cada nueva instancia en la clase obtiene una copia de las variables de instancia que esa clase define.
- Cada una puede después cambiar los valores sin afectar a ninguna otra variable.
- Con estas variables, sólo existe una copia de esa variable.
- Cada instancia de la clase tiene acceso a esa variable, pero existe sólo un valor. Al cambiarlo en ella lo hará para todas las instancias de esa clase.
- Para definir las variables de clase, se incluye antes de la variable la palabra clave static.
- Ejemplo:

```
class FamilyMember {

static String surname = Johnson;

String name;

int age;

. . . .

}
```

- Las instancias de la clase FamilyMember, tienen cada una sus propios valores para nombre y edad, pero la variable de clase surname tiene sólo un valor para todos los miembros de la familia.
- Si se cambia el valor de surname, se afectarán todas las instancias de esa clase.
- Para tener acceso a estas variables se usa la misma notación de punto que para las de instancia.
- Para obtener o cambiar el valor de una de clase, se puede usar la instancia o el nombre de la clase al lado izquierdo del punto.
- Las siguientes dos líneas de salida imprimen el mismo valor:

FamilyMember dad = new FamilyMember();

System.out.println(El apellido de la familia es: + dad.surname);

System.out.println(El apellido de la familia es: + FamilyMember.surname);

- Como al utilizar una instancia se puede cambiar el valor de una variable de clase, es fácil confundirse con las variables de clase y el origen de sus valores.
- Por esta razón, es una buena idea usar el nombre de la clase siempre que se refiera a una variable de clase, lo cual hace al código más fácil de leer y es más sencillo de depurar cuando se presentan resultados extraños.

LLAMADA A LOS METODOS

- Llamar a un método en los objetos es similar a referirse a sus variables de instancia.
- Las llamadas a los métodos también utilizan la notación de punto.
- El objeto cuyo método se llama, está al lado izquierdo del punto, y el nombre del método y sus argumentos están al lado derecho del punto:

myObject.methodOne(arg1, arg2, arg3);

- Todos los métodos deben tener paréntesis enseguida aun si el método no tiene argumentos:

myObject.methodNoArgs();

- A su vez, si el método que llamó resulta ser un objeto que cuenta con otros métodos, se los puede anidar como con las variables:

myObject.getClass().getName();

- También se pueden combinar llamadas a métodos anidados y también referencias a variables de instancia:

myObject.var.methodTwo(arg1, arg2);

- El método System.out.println() es un buen ejemplo de la anidación de variables y métodos.
- La clase System, parte del paquete java.lang, describe el comportamiento específico del sistema.
- System.out es una variable de clase que contiene una instancia de la clase PrintStream, que apunta a la salida estándar del sistema. Estas instancias cuentan con un método println() que imprime una cadena a ese flujo de salida.
- El siguiente ejemplo muestra algunos métodos definidos en la clase String.
- Las cadenas incluyen métodos para pruebas y modificación, similares a las bibliotecas de este tipo en otros lenguajes.

Class TestString {

public static void main (String args[ ]) {

String str = Hoy es la época de vacaciones;

System.out.println(La cadena es: + str);

```
System.out.println(La longitud de esta cadena es:  + str.length);

System.out.println(El carácter en la posición 5 es:  +

str.charAt(5));

System.out.println(La subcadena desde el carácter 11 hasta el 17

es:  + str.substring(11,17));

System.out.println(La posición del carácter v es:  +

str.indexOf(`v'));

System.out.print (El índice del inicio de la );

System.out.println(subcadena \época\ es:  +

str.indexOf(época));

System.out.println(La cadena en mayúsculas es:  +

str.toUpperCase());

}

}
```

Métodos de Clase

- Se aplican a la clase y no a sus instancias.
- Por lo común, son utilizados por los métodos de utilería que pueden no operar en forma directa en una instancia de esa clase, pero se adecuan a la clase de manera conceptual.
- Por ejemplo, la clase String contiene un método de clase llamado valueOf(), que puede tomar uno de varios tipos diferentes de argumentos (enteros, booleanos y otros objetos).
- Este método regresa entonces a una nueva instancia de String que contiene el valor de la cadena del argumento que se dio.
- Este método no opera de modo directo en una instancia existente de String, aunque obtener una cadena de otro objeto o tipo de dato es, en definitiva, una operación de cadena y tiene sentido definirla en la clase String.

- Los métodos de clase también pueden ser útiles para conjuntar métodos generales en un lugar (la clase).
- Por ejemplo, la clase Math, definida en el paquete java.lang, contiene un conjunto muy grande de operaciones matemáticas, como métodos de clase; no existen instancias de la clase Math, pero aún puede usar sus métodos con argumentos numéricos o booleanos.
- Para llamar a un método de clase, se utiliza la notación punto.
- Al igual que con las variables de clase, se puede usar una instancia de clase o la clase misma, a la izquierda del punto.
- Sin embargo, emplear el nombre de la clase para estas variables hace que su código sea más fácil de leer.
- Las últimas dos líneas en el siguiente ejemplo producen el mismo resultado:

```
String s, s2;

s = foo;

s2 = s.valueOf(5);

s2 = String.valueOf(5);
```

REFERENCIA A OBJETOS

- Al trabajar con objetos, algo importante que sucede es el uso de las referencias a dichos objetos.
- Cuando se asignan objetos a las variables, o se los pasa a los métodos como argumentos, se trasladan referencias a ellos, no los objetos mismos o las copias de éstos.
- Ejemplo:

```
import java.awt.Point;

class ReferencesTest {

public static void main (String args[ ] {

Point pt1, pt2;

pt1 = new Point(100,100);

pt2 = pt1;

pt1.x = 200;

pt1.y = 200;

System.out.println(Punto1:  + pt1.x + ,  + pt1.y);

System.out.println(Punto2:  + pt2.x + ,  + pt2.y);

}

}
```

- En este programa se declaran dos variables de tipo Point y se asigna un nuevo objeto de este tipo a pt1. Después se asigna el valor de pt1 a pt2.
- Cuando se asigna el valor de pt1 a pt2, de hecho se crea una referencia de pt2 al mismo objeto al cual se refiere pt1.
- Si se cambia el objeto al que se refiere pt2, también se modificará el objeto al que apunta pt1, ya que ambos se refieren al mismo objeto.

CAPITULO 5: ARREGLOS, CONDICIONALES Y CICLOS

5.1. ARREGLOS

- Los arreglos en Java son objetos que pueden pasarse y tratarse como otros objetos.
- Los arreglos son una forma de almacenar una lista de elementos.

- Cada espacio del arreglo guarda un elemento individual y se pueden colocar los elementos o cambiar el contenido de esos espacios según se necesite.
- Los arreglos pueden tener cualquier tipo de valor de elemento (tipos primitivos u objetos), pero no se pueden almacenar diferentes tipos en un solo arreglo.
- Es factible tener un arreglo de enteros o de cadenas o de arreglos, pero no se pueden tener arreglos de cadenas y enteros al mismo tiempo, por ejemplo.

CREACIÓN DE ARREGLOS EN JAVA

- Declarar una variable para guardar el arreglo.
- Crear un nuevo objeto de arreglo y asignarle a la variable de arreglo.
- Guardar lo que se desee en el arreglo.

DECLARACION DE VARIABLES DE ARREGLO

- Se declaran indicando el tipo de objeto que el arreglo contendrá y el nombre del arreglo, seguido por corchetes vacíos.

. Ejemplos:

String palabrasDifíciles[ ];

Point hits[ ];

int temps[ ];

- Un método alterno es colocar los corchetes después del tipo, en lugar de enseguida de la variable. Este método es más legible.

. Ejemplos:

String[ ] palabrasDifíciles;

Point[ ] hits;

int[ ] temps;

CREACION DE OBJETOS DE ARREGLO

- El segundo paso es crear un objeto de arreglo y asignarlo a esa variable.
- Existen dos formas de hacerlo:

Usar new

Inicializar de manera directa el contenido de ese arreglo.

- El primero implica el uso del operador new para crear una nueva instancia de un arreglo:

String[ ] nombres = new String[10];

- Esta línea crea un nuevo arreglo de Strings con diez slots (casillas) conteniendo los elementos.
- Cuando se crea un nuevo objeto de arreglo con new se deben indicar cuántas casillas tendrá ese

arreglo.
- Los objetos de arreglo pueden contener tipos primitivos o booleanos, de la misma forma que contienen objetos:

```
int[ ] temps = new int[99];
```

- Cuando se crea un objeto de arreglo mediante el uso de new, todas sus casillas se inicializan con algún valor: 0 para arreglos numéricos, false para booleanos, `\0' para arreglos de caracteres y null para objetos.
- También se puede crear e inicializar un arreglo al mismo tiempo.
- En lugar de utilizar new, se encierran los elementos del arreglo dentro de llaves, separados por comas:

```
String[ ] chiles = {jalapeno, anaheim, serrano, habanero, thai};
```

- Cada elemento dentro del arreglo dentro de las llaves debe ser del mismo tipo y debe coincidir con el tipo de la variable que contiene ese arreglo.
- Un arreglo del tamaño de elementos que se incluyeron se creará en forma automática; por ejemplo, en el caso anterior se crea un objeto String llamado chiles que contiene cinco elementos.

ACCESO A LOS ELEMENTOS DEL ARREGLO

- Para obtener un valor de un arreglo, se utiliza la expresión de arreglo subíndice:

```
arreglo[subíndice];
```

- subíndice especifica la casilla a consultar dentro del arreglo.
- Los subíndices (subscripts) de arreglos inician con 0, por lo que un arreglo con 10 elementos, tendrá diez casillas a las cuales se puede tener acceso al utilizar los subíndices del 0 al 9.
- Cuando se usan arreglos, se revisan todos los subíndices del arreglo para asegurarse de que estén dentro de su frontera: igual o mayor a cero y menor o igual a n−1, donde n es la longitud del arreglo, ya sean cuando se compila el programa o cuando se ejecuta.
- En Java no se puede tener acceso a asignar un valor a una casilla del arreglo fuera de las fronteras de éste. Ejemplo:

```
String[ ] arr = new String[10];

arr[10] = enunciado;
```

- Este último enunciado produce un error de compilación puesto que el arreglo arr sólo tiene 10 casillas, numeradas de 0 a 9. El elemento arr[10] no existe.
- Si el subíndice del arreglo se calcula al momento de la ejecución (por ejemplo, como parte de un ciclo) y está fuera de las fronteras del arreglo, el intérprete Java también produce un error, es decir, lanza una excepción.
- Para evitar cometer estos errores en forma accidental, se puede examinar la longitud del arreglo utilizando la variable de instancia length, que está disponible para todos los objetos de arreglo sin importar el tipo:

```
int len = arr.length // retorna el valor 10
```

ASIGNACION DE VALORES A ELEMENTOS DE UN ARREGLO

- Para asignar un valor de elemento a una casilla de un arreglo, se usa un enunciado de asignación:

```
arreglo[1] = 15;

sentencia[0] = El;

sentencia[10] = sentencia[0];
```

ARREGLOS MULTIDIMENSIONALES

- Java no soporta los arreglos multidimensionales directamente, pero permite declarar y crear un arreglo de arreglos, y esos arreglos pueden contener arreglos y así en los sucesivo, en tantas dimensiones como se necesite, y tener acceso a él como se haría en un arreglo multidimensional estilo C:

```
int coords[ ] [ ] = new int[12][12];

coords[0][0] = 1;

coords[0][1] = 2;
```

ENUNCIADOS DE BLOQUE

- Un enunciado de bloque es un grupo de otros enunciados rodeados por llaves ({}).
- Se puede usar un bloque en cualquier lugar a donde podría ir un enunciado individual, y el nuevo bloque crea otro nuevo propósito local para enunciados que están dentro de él.
- Esto significa que se pueden declarar y usar variables locales dentro un bloque y éstas dejarán de existir después de que termine la ejecución del bloque.
- En el siguiente ejemplo, se muestra un bloque dentro de una definición de método que declara una nueva variable y. No se puede emplear esa variable fuera del bloque donde se declaró:

```
void testblock() {

int x = 10;

{ // inicio del bloque

int y = 50;

System.out.println(dentro del bloque:);

System.out.println(x:  + x);

System.out.println(y:  + y);

} // fin del bloque

}
```

CONDICIONALES if

```
if (condicional)

enunciado
```

else

enunciado

Ejemplos:

```
if (x < y)

System.out.println(x es menor que y);

if (x < y)

System.out.println(x es menor que y);

else

System.out.println(y es menor que x);

if (engineState == true)

System.out.println(El motor ya está encendido.);

else {

System.out.println(Encendiendo el motor.);

if (gasLevel >= 1)

engineState = true;

else

System.out.println(Sin gasolina! No se enciende el motor.);

}
```

- Este ejemplo utiliza la prueba (engineState == true). Para pruebas booleanas de este tipo, una forma común es incluir la primera parte de la expresión en lugar de hacerlo en forma explícita y probar su valor contra verdadero o falso:

```
if (engineState)

System.out.println(El motor está encendido.);

else {

System.out.println(El motor está apagado.);
```

EL OPERADOR CONDICIONAL

- Una alternativa para utilizar las palabras clave if y else en un enunciado condicional es usar el operador condicional, también conocido como el operador ternario.

- El operador condicional es un operador ternario porque tiene tres términos.
- Este operador es una expresión, es decir regresa un valor (a diferencia del if más genérico que sólo puede resultar en un enunciado o bloque que se ejecute).
- El operador condicional es más útil para condicionales muy cortos o sencillos, y tiene este formato:

prueba ? resultadoverdadero : resultadofalso

- La prueba es una expresión que regresa true o false. Si la prueba es verdadera, el operador condicional regresará el valor de resultadoverdadero y si es falsa, regresará el valor de resultadofalso.

Ejemplo:

int smaller = x < y ? x : y;

- En este ejemplo, se prueban los valores de x e y, regresando el más pequeño de los dos y asignándolo a la variable smaller.

CONDICIONALES SWITCH

- Se los utiliza para evitar hacer if's anidados cuando se quiere comparar una variable contra algún valor.
- Formato:

switch (prueba) {

case valorUno:

sentenciaUno;

break;

case valorDos:

sentenciaDos;

break;

case valorTres:

sentenciaTres;

break;

. . . .

default: sentenciaDefault;

}

- Con esta estructura de programación, la prueba (que puede ser byte, char, short o int) se compara con cada uno de los valores en turno.
- Si es igual, se ejecuta la sentencia o sentencias después de la prueba; si no es igual, se ejecutará la

sentencia después de default. Este enunciado es opcional.

- Las pruebas y los valores sólo pueden ser de tipos primitivos y no se pueden usar tipos primitivos mayores, como long y float, cadenas u otros objetos.
- Tampoco pueden probarse otras relaciones más que las de igualdad, por lo que el uso de switch es limitado. Los if anidados pueden funcionar para cualquier clase de prueba en cualquier tipo.

Ejemplo:

```
switch (oper) {

case '+':

addargs(arg1, arg2);

break;

case '−':

subargs(arg1, arg2);

break;

case '*':

multargs(arg1, arg2);

break;

case '/':

divargs(arg1, arg2);

break;

}
```

- Si no se utiliza break en cada case, se ejecutarán los enunciados para cuando haya coincidencia con prueba y también todos los enunciados o sentencias inferiores en switch, hasta encontrar un break o hasta llegar al final de switch.
- Es preferible siempre, entonces, colocar break al final de cada enunciado, para hacer que sólo el enunciado verdadero se ejecute.
- Un uso adecuado en el cual no se requiere usar break, se da cuando se desea que múltiples valores ejecuten los mismos enunciados. Para esto se pueden utilizar muchas líneas en un caso sin ningún resultado, y switch ejecutará los primeros enunciados que encuentre.
- En el siguiente ejemplo, la cadena "x es un número par. " se imprimirá si x tiene valores de 2, 4, 6 u 8. Los otros valores imprimirán la cadena "x es un número impar."

```
switch (x) {

case 2:

case 4:
```

```
case 6:

case 8:

System.out.println("x es un número par.");

break;

default: System.out.println("x es un número impar.");
```

CICLOS for

- El ciclo for repite una declaración o bloque de enunciados un número de veces hasta que una condición se cumple.
- Estos ciclos con frecuencia se usan para una iteración sencilla en donde se repite un bloque de enunciados un cierto número de veces y después se detiene, aunque también se los puede usar para cualquier clase de ciclo.
- Formato:

```
for (inicialización; test; incremento) {

sentencias;

}
```

- El inicio del ciclo for tiene tres partes:
- **inicialización** es una expresión que inicializa el principio del ciclo. Si se tiene un índice, esta expresión puede ser declarada e inicializada, int i = 0. Las variables que se declaran en esta parte del ciclo son locales al ciclo: dejan de existir después de acabar la ejecución del mismo.
- **test** es la prueba que ocurre después de cada vuelta del ciclo. La prueba debe ser una expresión booleana o una función que regresa un valor booleano, por ejemplo, i < 10. Si la prueba es verdadera, el ciclo se ejecuta; cuando es falsa, el ciclo detiene su ejecución.
- **incremento** es una expresión o llamada de función. Por lo común, el incremento se utiliza para cambiar el valor del índice del ciclo a fin de acercar el estado del ciclo a *false* y que se complete.
- Dentro del ciclo for existen sentencias que se ejecutan cada vez que se itera el ciclo. Al igual que con if, se puede incluir un solo enunciado o un bloque; en este último caso, como se sabe, se lo empieza y termina con llaves ({ }).
- Ejemplo. ciclo for para inicializar todos los valores de un arreglo String para cadenas vacías:

```
String strArray [ ] = new String[10];

int i; // índice del lazo

for (i = 0; i < strArray.length; i++)

strArray[i] = "";
```

- Cualquier parte del ciclo for puede se un enunciado vacío, esto es, puede incluir sólo un punto y coma sin ninguna expresión o declaración, y esa parte del ciclo se ignorará.
- También puede tenerse un enunciado vacío para el cuerpo del ciclo for, si todo lo que se desea hacer se encuentra en la primera línea. Por ejemplo, aquí está una que encuentra el primer número primo mayor que 4000:

```
for (i = 40001; notPrime(i); i +=2)

;
```

CICLOS while y do

- Estos ciclos permiten también a un bloque de código ejecutarse de manera repetida hasta encontrar una condición específica. Utilizar uno de estos tres ciclos es, por lo general, una cuestión de estilo de programación.
- Los ciclos while y do son exactamente los mismos que en C y C++, a excepción de que su condición de prueba debe ser un booleano.

CICLOS while

- Se utilizan para repetir un enunciado o bloque de enunciados, mientras una condición sea verdadera.
- Formato:

```
while (condición) {

cuerpoDelLazo;

}
```

- La condición es una expresión booleana. Si regresa true, el ciclo while ejecuta los enunciados en cuerpoDelLazo y después prueba la condición hasta que ésta sea falsa. También se puede emplear un enunciado sencillo en lugar del bloque.
- Ejemplo. Inicialización y escritura de un vector:

```
int [ ] A = new int[10];

for (int j = 0; j < A.length; j++)

A[ j ] = j + 2;

int i = 0;

while (i < A.length)

System.out.println("A[ " + i + " ] = " + A[i]);
```

- Si la condición la primera vez que se prueba es falsa, el cuerpo del ciclo while nunca se ejecutará.
- Para este caso, cuando se desee que una iteración se ejecute por lo menos una vez, se utiliza el ciclo do ... while, como se explica a continuación.

CICLOS do ... while

- El ciclo do es como el ciclo while, excepto que ejecuta un enunciado o bloque dado hasta que la condición es false.
- La principal diferencia es que los ciclos while prueban la condición antes de realizar el ciclo, lo cual hace posible que el cuerpo del ciclo nunca se ejecute si la condición es falsa la primera vez que se prueba.
- Los ciclos do ejecutan el cuerpo del ciclo por lo menos una vez antes de probar la condición.

- Formato:

```
do {

cuerpoDelLazo;

} while (condición);
```

- cuerpoDelLazo se ejecuta por lo menos una vez.
- La condición se prueba al final de la ejecución del cuerpo.
- La condición es una prueba booleana: si regresa true, el ciclo se ejecuta otra vez; si regresa false, se detiene.

Ejemplo:

```
int x = 1;

do {

System.out.println("Lazo, vuelta " + x);

x++;

} while (x <= 10);
```

SALIDA DE CICLOS

- Todos los ciclos (for, while y do) se terminan cuando la condición que prueba se cumple.
- Para salir del ciclo de manera incondicional, se usan las palabras clave break y continue.
- break detiene la ejecución del ciclo actual. Si se anidaron ciclos, la ejecución selecciona el siguiente ciclo externo; de otra manera, el programa continúa la ejecución del siguiente enunciado después del ciclo.
- Ejemplo: copia de elementos de un arreglo de números enteros a un arreglo de punto flotante hasta el final del arreglo o hasta que se encuentre un 0.

```
int count = 0;

while (count < array1.length) {

if (array1[count] ==0) {

break;

}

array2[count] = (float) array1[count++];

}
```

- El uso de continue es similar al de break, a excepción de que en lugar de detener por completo la ejecución del ciclo, éste inicia otra vez la siguiente iteración.
- Para los ciclos do y while, esto significa que la ejecución del bloque se inicia de nuevo.

- Para los ciclos for, la expresión de incremento se evalúa y después el bloque se ejecuta.
- continue es útil cuando se desea tener elementos especiales de condición dentro de un ciclo.
- Usando el ejemplo anterior, se puede probar si el elemento actual es 0 y reiniciar el ciclo si lo encuentra, para que el arreglo resultante nunca contenga cero. Obsérvese que puesto que en el primer arreglo se saltan los elementos, ahora se tiene que seguir a dos diferentes contadores de arreglos:

```
int count1 = 0;

int count2 = 0;

while (count1 < array1.length) {

if (array1[count1] ==0) {

count1++;

continue;

}

array2[count2++] = (float) array1[count1++];

}
```

CICLOS ETIQUETADOS

- Tanto break como continue pueden tener una etiqueta opcional que le indique a Java dónde detenerse.
- Sin una etiqueta, break saltará hacia el ciclo más cercano (a uno envolvente o al siguiente enunciado fuera del ciclo) y continue reinicia el ciclo envolvente.
- Utilizar estas instrucciones etiquetadas, permite salirse de ciclos anidados o continuar un ciclo fuera del actual.
- Para utilizar un ciclo etiquetado, se agrega la etiqueta antes de la parte inicial del mismo, con dos puntos entre ellos.
- Después, cuando se utilice break o continue, se agrega el nombre de la etiqueta, después de la palabra clave:

```
out:

for (int i = 0; i < 10; i++) {

while (x < 50) {

if (i * x == 400)

break out;

. . .

}

. . .
```

}

- En este recorte de código, la etiqueta out marca el ciclo externo. Entonces, dentro de los ciclos for y while, cuando una condición particular se cumple, un break ocasiona que la ejecución se salga de ambos ciclos.
- El siguiente programa contiene un ciclo anidado for. Dentro del ciclo más profundo, si los valores sumados de los dos contadores es mayor que cuatro, ambos ciclos se abandonan al mismo tiempo:

```
foo:

for (int i = 1, i <= 5; i++)

for (int j = 1; j <= 3; j++) {

System.out.println(" i es " + i + ", j es " + j);

if (i + j) > 4)

break foo;

}

System.out.println("fin de lazos");
```

Salida del programa:

```
i es 1, j es 1

i es 1, j es 2

i es 1, j es 3

i es 2, j es 1

i es 2, j es 2

i es 2, j es 3

fin de lazos
```

- Como se puede ver, el ciclo se iteró hasta que la suma de i y j fue mayor que 4 y después ambos regresaron al bloque externo y el mensaje final se imprimió.

CAPITULO 6: CREACION DE CLASES Y APLICACIONES EN JAVA

Objetivos:

- Definir clases
- Declarar y usar variables de instancia
- Definir y usar métodos
- Crear aplicaciones Java, incluido el método main() y cómo pasar argumentos a un programa Java desde una línea de comandos

5.1 Definición de clase

- Se usa la palabra reservada class y el nombre de la clase:

```
class NombreDeClase {

....

}
```

- Si esta clase es una subclase de otra, utilice extends para indicar su superclase:

```
class nombreDeClase extends nombreSuperClase {

....

}
```

- Si esta clase implementa una interfaz específica, utilice implements para referirse a ella:

```
class NombreClaseEjecutora implements Corredora {

....

}
```

- Tanto extends como implements son opcionales.

CREACION DE VARIABLES DE INSTANCIA Y DE CLASE

- Se declaran fuera de la definición de un método de clase.
- Normalmente se definen después de la primera línea de la definición de clase.
- Ejemplo:

```
class Bicicleta extends VehículoManejadoPorPersonas {

String bikeType;

int chainGear;

int rearCogs;

int currentGearFront;

int currentGearRear;

}
```

- Para este ejemplo se tienen cuatro variables de instancia:
- bikeType: indica la clase de bicicleta, por ejemplo de montaña o de calle.
- chainGear, el número de velocidades en la parte delantera.
- rearCogs, el número de velocidades menores en el eje trasero.

- currentGearFront y currentGearRear: las velocidades actuales de la bicicleta están tanto en el frente como en la parte trasera.

CONSTANTES

- Las constantes son útiles para definir valores comunes para todos los métodos de un objeto (dar nombres significativos a valores de objetos generales que nunca cambiarán).
- En Java se pueden crear constantes sólo para variables de instancia o de clase, no para variables locales.
- Una variable constante o constantes es un valor que nunca cambia.
- Para declara una constante, se utiliza la palabra clave **final** antes de la declaración de la variable y se incluye un valor inicial para esa variable:

final float pi = 3.141592;

final boolean debug = false;

final int maxsize = 40000;

VARIABLES DE CLASE

- Son globales para todas sus instancias de clase.
- Son usadas para la comunicación entre diferentes objetos con la misma clase, o para rastrear los estados globales de entre una serie de objetos.
- Se usa la palabra reservada **static**.

static int sum;

static final int maxObjects = 10;

CREACION DE METODOS

- Los métodos son usados para definir el comportamiento de un objeto.

Definición de métodos

- Constan de cuatro partes básicas:
- El nombre del método
- El tipo de objeto o tipo primitivo que el método regresa.
- Una lista de parámetros
- El cuerpo del método

- En Java se puede tener más de un método con el mismo nombre, pero con un tipo de retorno o lista de argumentos diferente.

Formato:

tipoderetorno nombre (tipo1 arg1, tipo2 arg2, tipo3 arg3, ...) {

....

}

- El tipoderetorno es el tipo primitivo o clase del valor que este método regresa. Puede ser uno de los tipos primitivos, un nombre de clase o void si no regresa un valor.
- Si el método regresa un objeto de arreglo, las llaves del arreglo van después del tipo de retorno o después de la lista de parámetros: se usará la primera forma porque es más fácil de leer:

```
int [ ] hacerRango (int lower, int upper) { ... }
```

- La lista de parámetros del método es un conjunto de declaraciones de variables separadas por comas dentro de los paréntesis.
- Estos parámetros se vuelven variables locales en el cuerpo del método, cuyos valores son los objetos o los valores primitivos transmitidos cuando se llama al método.
- Dentro del cuerpo de un método se pueden tener enunciados, expresiones, llamadas de métodos a otros objetos, condicionales, ciclos y demás.
- Si el método tiene un tipo de retorno, en algún lugar dentro del cuerpo del método se necesitará regresar un valor. Se debe usar la palabra clave **return** para hacer esto.
- Ejemplo: método que toma dos enteros (un límite inferior y uno superior) y crea un arreglo que contiene todos los enteros entres los dos límites, incluidos los límites:

```
class RangeClass {

int [ ] makeRange (int lower, int upper) {

int arr [ ] = new int [ (upper . lower) + 1 ];

for (int i = 0; i < arr.length; i++) {

arr [i ] = lower++;

}

return arr;

}

public static void main (String args[ ] ) {

int theArray[ ];

RangeClass theRange = new RangeClass();

theArray = theRange.makeRange(1,10);

System.out.print(El arreglo: [ );

for (int i = 0; i < theArray.length; i++) {

System.out.print(theArray [i ] +  )

}

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

```
}

}
```

- Salida del programa:

```
The array: [ 1 2 3 4 5 6 7 8 9 10 ]
```

- El método main() en esta clase prueba el método makeRange() al crear un arreglo donde los límites inferior y superior del rango son 1 y 10, respectivamente y después utiliza un ciclo for a fin de imprimir los valores del nuevo arreglo.

PALABRA CLAVE this

- Es usada para referirse al objeto actual, aquel objeto que el método llamó.
- También se usa para referirse a las variables de instancia de ese objeto o para pasar el objeto actual como un argumento a otro método.
- Ejemplo:

```
t = this.x // variable de instancia x para este objeto

this.myMethod(this) // llama al método myMethod, definido en esta clase y le

// pasa el objeto actual

return this; // retorna el objeto actual
```

- También es factible referirse a las variables de instancia y a las llamadas de método definidas en clase actual sólo por el nombre; this en este caso está implícito.
- Los primeros ejemplos anteriores podrían escribirse de esta manera:

```
t = x;

myMethod(this)
```

- Ya que this es una referencia a la instancia actual de una clase, se la debe usar sólo dentro del cuerpo de una definición de método de instancia. Los métodos de clase no emplean this.

ALCANCE DE LA VARIABLE Y DEFINICIONES DE METODO

- Cuando se refiere a una variable dentro de sus definiciones de método, Java busca una definición de esa variable primero en el ámbito actual (un bloque por ejemplo), después en el exterior hasta la definición del método actual.
- Si esa variable no es local, entonces Java busca una definición de ella como instancia o variable de clase en la clase actual, y por último, en cada superclase en turno.
- De esta manera, es posible crear una variable en un ámbito más corto de tal forma que una definición de esa misma variable oculte el valor original de aquella variable.
- Esto introduce errores sutiles y confusos en el código.
- Ejemplo:

```
class ScopeTest {
```

```
int test = 10;

void printTest(){

int test = 20;

System.out.println(test =  + test);

}

}
```

PASO DE ARGUMENTOS A LOS METODOS

- En el paso de objetos como argumentos, las variables se pasan como referencias, lo que implica que lo que se les haga a esos objetos dentro del método afectará los objetos originales también.
- Esto incluye a los arreglos y a todos los objetos que los arreglos contienen.
- Cuando se pasa un arreglo a un método y modifica su contenido, el arreglo original se ve afectado.
- Los tipos de datos primitivos, en cambio, se pasan por valor.
- Ejemplo:

```
class PassByReference {

int OnetoZero (int arg[ ]) {

int count = 0;

for (int i = 0; i < arg.lenth; i++) {

if (arg [ i] == 1) {

count++;

arg [ i] = 0;

}

}

return count;

}

}
```

- El método onetoZero realiza dos acciones:
- Cuenta el número de unos en el arreglo y regresa ese valor.
- Si encuentra un uno, lo sustituye por un cero en el arreglo.
- El siguiente programa muestra el método main() para la clase PassByReference, el cual prueba el método onetoZero:

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

```
int arr[ ] = { 1,3,4,5,1,1,7 };

PassByReference test = new PassByReference();

int numOnes;

System.out.print(Valores del arreglo: [ );

for (int i = 0; i < arr.length; i++) {

System.out.print(arr};

}

Sytem.out.println( ] );

numOnes = test.OnetoZero(arr);

System.out.println(Número de Unos =  + numOnes);

System.out.print(Nuevos valores del arreglo: [ );

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[ i] +  );

}

System.out.println( ] );

}
```

Salida del programa:

Valores del arreglo: [1 3 4 5 1 1 7]

Número de unos = 3

Nuevos valores del arreglo: [0 3 4 5 0 0 7]

METODOS DE CLASE

- Los métodos de clase están disponibles para cualquier instancia de la clase misma así como en otras clases. En consecuencia, algunos métodos de clase se usan en cualquier lugar, sin importar si una instancia de la clase existe o no.
- Para definir los métodos de clase, se emplea la palabra clave static al frente de la definición del método.

```
static int max (int arg1, int arg2)
```

COMO CREAR APLICACIONES JAVA

- Las aplicaciones son programas Java ejecutados por sí mismos.
- Son diferentes a los applets, los cuales requieren HotJava o un visualizador con capacidad Java para que puedan verse.
- Una aplicación Java consiste en una o más clases. Para ejecutar una aplicación Java se requiere de una clase que funcione como punto de arranque para el resto del programa.
- La clase de arranque requiere de un método main().
- Cuando ejecuta la clase Java compilada (mediante el intérprete Java), el método main() es lo primero que se llama.
- El método main() siempre debe ser hecho así:

public static void main (String args [ ])

- public significa que este método estará disponible para otras clases y objetos, por lo que debe declararse como público al método main().
- static significa que es un método de clase
- void indica que no regresa valores el método
- main() toma un parámetro: un arreglo de caracteres. Se usa para argumentos de línea de comando.
- El cuerpo contiene cualquier código que se requiera para que la aplicación comience: inicializar variables o crear instancias de cualquier clase que se hayan declarado.

APLICACIONES JAVA Y ARGUMENTOS DE LA LINEA DE COMANDO

- Ya que las aplicaciones Java son programas individuales, es útil poder pasar argumentos u opciones a éstos para determinar cómo se va a ejecutar o permitir a un programa genérico operar con diferentes clases de entradas.
- Los argumentos de la línea de comando se usan para activar entradas de depuración, indicar un nombre de archivo desde el cual lea o escriba o para cualquier otra información que se desee pasar a un programa.

Paso de argumentos a los programas Java

- Para pasar argumentos sólo hay que agregarlos a la línea de comando cuando se lo ejecute:

java Miprograma argumentoUno 2 tres

- Esta línea de comando tiene tres argumentos: argumentoUno, el número 2 y tres. Un espacio separa los argumentos.

Java Miprograma Java is cool

- Para unir argumentos, se los debe encerrar entre comillas dobles:

Java Miprograma Java is cool

- Las comillas dobles desaparecen antes de que el argumento llegue al programa Java.

Manejo de argumentos en los programas Java

- Java guarda los argumentos en un arreglo de cadenas, el cual se pasa al método main() en el programa Java. El método luce así:

public static void main (String args[ ]) { ... }

- Aquí, args es el nombre del arreglo de cadenas que contiene la lista de argumentos.
- Dentro del método main() se pueden manejar los argumentos dados al programa al hacer iteraciones sobre el arreglo de argumentos y manejar esos argumentos como se desee.
- El siguiente ejemplo es una clase que imprime los argumentos que obtiene, uno por línea.

```
class EchoArgs {

public static void main(String args[ ]) {

for (int i = 0; i < args.length; i++) {

System.out.println(Argumento  + i + :  + args[ i ]);

}

}

}
```

- La siguiente es una entrada y salida de muestra de este programa:

```
java EchoArgs 1 2 3 jump

Argumento 0: 1

Argumento 1: 2

Argumento 2: 3

Argumento 3: jump

java EchoArgs foo bar zap twaddle 5

Argumento 0: foo bar

Argumento 1: zap

Argumento 2: twaddle

Argumento 3: 5
```

- Los argumentos que se pasan se almacenan en un arreglo de cadenas, por lo que cualquier argumento que se pase a un programa Java serán cadenas guardadas en el arreglo de argumentos.
- Para no tratarlos como cadenas, se los deberá convertir a cualquier tipo que se quiera que sean.
- Ejemplo: programa SumAverage que toma cualquier número de argumentos numéricos y regresa la suma y el promedio de esos argumentos:

```
class SumAverage {

public static void main (String args[ ]) {

int suma = 0;
```

```
for (int i = 0; i < args.length; i++)

suma += Integer.parseInt(args[ i ]);

System.out.println(La suma es:  + suma);

System.out.println(El promedio es:  + (float)sum / args.length);

}

}
```

- Se usa el método de la clase Integer llamado parseInt para pasar cadenas de caracteres a enteros.
- Si no se usa este método saldrá un mensaje de error como el siguiente:

SumAverage.java:9: Incompatible type for +=. Can't convert java.lang.String to int.

suma += args [i];

- Si lo ejecutamos con:

java SumAverage 1 2 3 regresa la siguiente salida:

La suma es: 6

El promedio es: 2

METODOS EN JAVA

- Los métodos son la parte más importante de cualquier lenguaje orientado a objetos, pues mientras las clases y los objetos ofrecen la estructura, y las variables de clase y de instancia una manera de contener los atributos y el estado de esa clase u objeto, los métodos son los que en realidad determinan el comportamiento del objeto y definen cómo interactúa con otros en el sistema.

CREACION DE METODOS CON EL MISMO NOMBRE Y ARGUMENTOS DIFERENTES

- En Java se pueden sobrecargar (overloading) métodos con el mismo nombre, pero con diferentes identificaciones y definiciones.
- La sobrecarga de métodos permite a las instancias de la clase tener una interfaz más sencilla para otros objetos, sin necesidad de métodos por completo diferentes que realizan en esencia lo mismo.
- La sobrecarga permite comportarse a los métodos en forma distinta en base a la entrada a ese método.
- Cuando se llama a un método en un objeto, Java hace coincidir su nombre, número y tipo de argumentos para seleccionar qué definición de método ejecutar.
- Para crear un método sobrecargado, se deben crear varias definiciones distintas de método en la clase, todas con el mismo nombre, pero con diferentes listas de parámetros, ya sean en número o tipo de argumentos.
- Java permite esto siempre que cada lista de parámetros sea única para el mismo nombre del método.
- Si se crean dos métodos con el mismo nombre, la misma lista de parámetros, pero diferentes tipos de retorno, se obtendrá un error de compilación.
- Los nombres de variables son irrelevantes, lo que importa es el número y el tipo.
- Ejemplo: se usará una definición para una clase MyRect que define una forma rectangular. Esta clase tiene cuatro variables de instancia para definir las esquinas del rectángulo: x1, y1, x2, y2.

```
class MyRect {

int x1 = 0;

int y1 = 0;

int x2 = 0;

int y2 = 0;

}
```

- Al instanciar esta clase, todas las variables de instancia se inicializan en cero.
- Ahora definamos un método buildRect() que toma 4 argumentos enteros y redimensiona el rectángulo para que tenga los valores apropiados en sus esquinas, devolviendo el objeto resultante.
- En este ejemplo, ya que los argumentos tienen los mismos nombres de las variables de instancia, debe usarse this para referirse a ellos:

```
MyRect buildRect (int x1, int y1, int x2, int y2) {

this.x1 = x1;

this.y1 = y1;

this.x2 = x2;

this.y2 = y2;

return this;

}
```

- Para definir las dimensiones del rectángulo de manera diferente, por ejemplo, utilizando objetos Point, se puede sobrecargar el método buildRect() para que su lista de parámetros tome dos objetos de este tipo.
- Para esto se necesita importar la clase Point al inicio del archivo fuente para que Java pueda encontrarla.

```
MyRect buildRect (Point topLeft, Point bottomRight) {

x1 = topLeft.x;

y1 = topLeft.y;

x2 = bottomRight.x;

y2 = bottomRight.y;

return this;

}
```

- Si se desea definir el rectángulo con el uso de una esquina superior y una anchura y una altura, sólo se crea una distinta definición para buildRect():

```
MyRect buildRect (Point topLeft, int w, int h) {

x1 = topLeft.x;

y1 = topLeft.y;

x2 = x1 + w;

y2 = y1 + h;

return this;

}
```

- Para finalizar, se creará un método para imprimir las coordenadas y un método main() para probarlo.
- El listado del programa MyRect.java muestra la definición de clase completa con todos sus métodos.

METODOS CONSTRUCTORES

- Además de los métodos ordinarios, también pueden definirse métodos constructores en las definiciones de clase.
- Un *método constructor* es un tipo especial de método que determina cómo se inicializa un objeto cuando se crea.
- A diferencia de los métodos ordinarios, no se puede llamar a un método constructor en forma directa; en su lugar, Java los llama de manera automática cuando se usa new para crear una nueva instancia de una clase.
- Al usar new se realizan tres acciones:

- Se asigna memoria para el objeto
- Se inicializan las variables de instancia de ese objeto, ya sea con sus valores iniciales o por el valor predeterminado: 0 para números, null para objetos, false para booleanos, `\0' para caracteres).
- Se llama al método constructor de la clase, que puede ser uno o varios.

- Si la clase no cuenta con algún método constructor especial definido, se crea el objeto con sus valores predeterminados.
- En estos casos, puede ser que se tenga que configurar sus variables de instancia o llamar a otros métodos que ese objeto necesita para inicializarse.
- Al definirse métodos constructores en sus propias clases, se pueden configurar valores iniciales de las variables de instancia, llamar métodos basados en esas variables o llamar métodos de otros objetos, o sino calcular propiedades iniciales del objeto.
- También es posible sobrecargar constructores, como si fueran métodos ordinarios, para crear un objeto que tiene propiedades específicas basadas en los argumentos que se le de a new.

CONSTRUCTORES BASICOS

- Los constructores son parecidos a los métodos ordinarios, con dos diferencias básicas:
- Siempre tienen el mismo nombre de la clase.
- No tienen un tipo de retorno (return).
- En el archivo Persona.java se muestra un ejemplo de una clase Persona con un constructor que

inicializa sus variables de instancia con base en los argumentos de new.

LLAMADA A OTRO CONSTRUCTOR

- Algunos constructores podrían ser un conjunto más amplio de otro constructor definido en la misma clase, es decir, pueden tener el mismo comportamiento y un poco más.
- En lugar de duplicar un comportamiento idéntico en múltiples métodos constructores en la clase, tiene sentido estar habilitado para sólo llamar al primer constructor desde dentro del cuerpo del segundo.
- Para hacer esto, se debe llamar al constructor definido en la clase actual de la siguiente manera:

this(arg1, arg2, arg3 ...);

- Los argumentos de this deben ser los del constructor.
- Ejemplo:

```
Persona(String n) {

int ed = 3;

nombre = n;

this(ed);

}

Persona(int a) {

edad = a;

}

public static void main (String args [ ]) {

Persona p;

p = new Persona(Laura);

p.printPersona();

System.out.println(−−−−−−−−);

}
```

SOBRECARGA DE CONSTRUCTORES

- Los constructores pueden también tomar varios números y tipos de parámetros, lo que permite crear exactamente un objeto con las propiedades que se desee, o para que pueda calcular propiedades a partir de distintas formas de entrada.
- Ejemplo: en el listado MyRect2.java se encuentran varios constructores, definidos de la misma manera que los métodos buildRect() de la clase MyRect, que constan en el archivo MyRect.java.

METODOS SOBREPUESTOS

- Cuando se llama a un método de objeto, se busca su definición en la clase de ese objeto, luego se pasa a una jerarquía mayor hasta encontrar su definición.
- La herencia permite definir y utilizar métodos de forma repetida en las subclases sin tener que duplicar el código.
- Cuando se desee que un objeto responda a los mismos métodos, pero que tenga diferente comportamiento cuando llame a un método, se puede sobreponer.
- Esto implica definir un método en una subclase que tiene la misma identificación de otro en una superclase.
- Cuando se llama al método, el de la subclase se busca y se ejecuta, en lugar del que está en la superclase.
- Para sobreponer un método, se debe crear uno en su subclase que tenga la misma identificación (nombre, tipo de retorno, lista de parámetros) de otro definido en una de sus superclases.
- Puesto que se ejecuta la primera definición del método que se encuentre y que coincida con la identificación, el método definido en la subclase oculta la definición del original.
- Ejemplo: en el listado PrintSubClass.java se muestra un método printMe() definido en la superclase y el listado PrintSubClass2.java que en cambio oculta el método printMe().

LLAMADA AL METODO ORIGINAL

- En general, existen dos razones para sobreponer un método que una superclase ya implementó:
- Sustituir la definición de ese método original por completo.
- Agrandar el método original, con comportamiento adicional.
- A veces se puede desear agregar un comportamiento a la definición original de un método, en vez de borrarla por completo.
- Esto es en particular útil cuando se duplica el comportamiento tanto en el método original como en el método que lo sobrepone.
- Al poder llamar al método original en el cuerpo del método que lo sobrepone, se puede adicionar únicamente lo que se necesite.
- Para llamar al método original desde dentro de una definición de método, se debe usar la palabra clave **super** para pasar la llamada al método hacia una jerarquía mayor:

```
void myMethod (String a, String b) {

// colocar código aquí

super.myMethod(a,b);

// colocar más código

}
```

- La palabra clave super, parecida a la palabra clave this, es un marcador para esta superclase de la clase.
- Se la puede usar en cualquier lugar en que se emplea this para referirse a la superclase en lugar de la clase actual.
- En el listado PrintSubclass3.java se encuentra un ejemplo que permite llamar al método original desde el método printMe() de la subclase PrintSubClass3.

SOBREPOSICION DE CONSTRUCTORES

- En términos técnicos, los constructores no se pueden sobreponer.
- Puesto que siempre tienen el mismo nombre de la clase actual, siempre se crearán constructores

nuevos en lugar de heredar los que ya se tenían.

- Esto en casi todas las ocasiones será adecuado, ya que cuando el constructor de una clase es llamado, otro con la misma identificación para todas sus superclases también es llamado, así que la inicialización de todas las partes de una clase que hereda puede llevarse a cabo.
- Sin embargo, cuando se definen constructores para una clase, tal vez se desee cambiar la inicialización de un objeto, no sólo al establecer los valores de las nuevas variables que la clase agrega, sino también para cambiar el contenido de las que ya están ahí.
- Para hacer esto, se pueden llamar explícitamente a los constructores de su superclase y después cambiar lo que se desee.
- Para llamar a un método ordinario en una superclase, se utiliza super.nombreMétodo(argumentos).
- Sin embargo, como los constructores no cuentan con un nombre de método parra llamarlo, se debe utilizar una forma diferente:

```
super(arg1, arg2, ... );
```

- Similar al uso de this (...) en un constructor, super(...) llama al método constructor de la superclase inmediata, la cual puede, a su vez, llamar al constructor de su superclase y así sucesivamente.
- En el listado NamedPoint.java se muestra una clase llamada NamedPoint, la cual extiende la clase Point del paquete awt de Java.
- Esta última clase tiene sólo un constructor, el cual toma un argumento x y uno y, y regresa un objeto Point.
- NamedPoint tiene una variable de instancia adicional (una cadena para el nombre) y define un constructor para inicializar a x, y, y el nombre.

METODOS DE FINALIZACION

- Son llamados justo antes de que un objeto sea recolectado como basura y reclame memoria.
- El método de finalizar es finalize().
- La clase Object define un método de finalización predeterminado, el cual no realiza nada.
- Para crear uno para las clases que se construyen, se lo debe sobreponer usando la identificación:

```
void finalize() {

.....

}
```

- Dentro del cuerpo del método finalize() se puede incluir cualquier instrucción adicional para ese objeto.
- También se puede llamar a super.finalize() para permitir a sus superclases terminar el objeto, si es necesario.
- El método se lo puede llamar en cualquier instante, pero llamarlo no provoca que un objeto sea llamado para recolectar basura.
- Sólo si se eliminan todas las referencias a un objeto, causará que sea marcado para borrarlo.

MODIFICADORES DE ACCESO

- Son prefijos que pueden aplicarse en varias combinaciones a los métodos y variables dentro de una clase y algunos a la clase misma.
- Los modificadores de acceso pueden ser public, protected o private.
- Todos estos son opcionales.

Control de acceso de variables y métodos

- El control de acceso se lleva a cabo sobre el control de la visibilidad.
- Cuando un método o variable es visible para otra, sus métodos pueden referenciarse (llamar o modificar) a ese método o variable.
- Para proteger un método o variable de estas referencias, se utilizan los cuatro niveles de visibilidad que se describen a continuación.
- Cada uno es más restrictivo que otro y por tanto, ofrecen mayor protección que el anterior a éste.
- Las protecciones pueden ser: public, package, protected y private.

public

- Cualquier método o variable es visible para la clase en la cual está definido.
- Para hacerlos visibles a todas las clases fuera de esa clase, se debe declarar al método o variable como public. Este es el acceso más amplio posible.
- Ejemplos:

```
public class ApublicClass {

public int aPublicInt;

public String aPublicString;

public float aPuclicMethod () {

. . .

}

}
```

- Una variable o método con acceso public tiene la visibilidad más amplia posible. Cualquier persona puede verla y utilizarla.
- No siempre puede desearse esto, lo cual lleva al siguiente nivel de protección.

package

- Los paquetes pueden agrupar clases.
- El siguiente nivel de protección se lo hace justamente a nivel de paquete, esto es, entre clases asociadas entre sí, dentro de un sistema, una biblioteca o programa (o cualquier otra agrupación de clases relacionadas).
- Este nivel de acceso no tiene un nombre preciso, pero está indicado por la ausencia de algún modificador de acceso en una declaración.
- Algunas veces ha sido llamado con varios nombres sugerentes, incluidos amigable y paquete. Este último parece ser el más apropiado y es el que utilizaremos aquí.
- Las declaraciones hasta hoy utilizadas en este curso son de este tipo y lucen así:

```
public class AlessPublicClass {

int aPackageInt = 2;

String aPackageString = a 1 and a ;
```

```
float aPackageMethod() { // no colocar modificador significa package

. . .

}

}

public class AclassInTheSamePackage {

public void testUse(){

AlessPublicClass aLPC = new AlessPublicClass();

System.out.println(aLPC.aPackageString + aLPC.aPackageInt);

ALPC.aPackageMethod();

}

}
```

- package se convirtió en el nivel de protección predeterminado porque cuando se diseña un gran sistema, se suelen dividir las clases en grupos de trabajo para implantar partes más pequeñas, y las clases con frecuencia necesitan compartir más entre ellas que con el mundo exterior.
- Si se desean detalles de implantación que no se quiere compartir, se tiene que usar otro nivel de protección.

protected

- Esta relación es entre una clase y sus subclases presentes y futuras. Estas subclases están más cerca de una clase padre que a cualquier clase externa por las siguientes razones:
- Las subclases son por lo común más conscientes de las partes internas de una clase padre.
- Las subclases con frecuencia están escritas por el programador o por alguien a quien se le dio el código fuente.
- Las subclases con frecuencia necesitan modificar o mejorar la representación de la información dentro de una clase padre.
- Nadie más tiene el privilegio en este nivel de acceso.
- Otras clases deberán conformarse con la interfaz pública que la clase presente.
- Este nivel ofrece mayor protección y reduce aún más la visibilidad, pero todavía les permite a las subclases un acceso total.
- Ejemplo:

```
public class AProtectedClass {

private protected int aProtectedInt = 4;

private protected String aProtectedString = and a 3 and a ;

private protected float aProtectedMethod() {

. . . .
```

```
}

}

public class AprotectedClassSubclass extends AprotectedClass {

public void testUse() {

AProtectedClassSubclass aPCs = new AprotectedClassSubclass();

System.out.println(aPCs.aProtectedString + aPCs.aProtectedInt);

aPCs.aProtectedMethod(); // todo esto está correcto

}

}

public class AnyClassInTheSamePackage {

public void testUse() {

AProtectedClassSubclass aPCs = new AprotectedClassSubclass();

System.out.println(aPCs.aProtectedString + aPCs.aProtectedInt);

aPCs.aProtectedMethod(); // nada está correcto

}

}
```

- A pesar de que AnyClassInTheSamePackage se encuentra en el mismo paquete que AprotectedClass, no es una subclase de ella (es una subclase de Object).
- Sólo a las subclases se les permite ver y utilizar variables y métodos privados *protected*.
- Las declaraciones en AprotectedClass tienen el prefijo de *private protected*, ya que en Java 1.0 agregar privado es necesario para obtener el comportamiento descrito.
- *protected* permite el acceso a las subclases y clases en el mismo paquete, con lo cual se ofrece un (quinto) nivel de protección combinado.
- Uno de los más claros ejemplos de la necesidad de usar este nivel especial de acceso es cuando se respalda una abstracción pública con una clase.
- En lo referente a la parte externa de la clase, se tiene una interfaz pública y sencilla (mediante métodos) a cualquier abstracción que se haya construido.
- Una representación más compleja y la implantación que depende de ella se oculta dentro.
- Cuando las subclases extienden y modifican esta representación, o incluso sólo su implantación, necesitan llegar a la representación fundamental y concreta y no sólo a la abstracción:

```
public class SortedList {

private protected BinaryTree theBinaryTree;

.....
```

```
public Object [ ] theList {

return theBinaryTree.asArray();

}

public void add(Object o) {

theBinaryTree.addObject(o);

}

}

public class InsertSortedList extends SortedList {

public void insert(Object o, int position) {

theBinaryTree.insertObject(o, position);

}

}
```

- Sin la capacidad de tener acceso a theBinaryTree, el método insert() habría obtenido la lista como un arreglo de Objects, mediante el método público theList(), así como distribuir un nuevo arreglo, más grande, e insertar el objeto nuevo en forma manual.
- Al ver que su padre utiliza un BinaryTree para implantar una lista ordenada, puede llamar al método insertObject() integrado en él par cumplir con su función.
- En Java, protected resuelve sólo una parte del problema, al permitir separar lo concreto de lo abstracto; el resto depende del programador.

private

- Es el nivel de protección más alto y con menor visibilidad que se puede obtener (opuesto totalmente a public).
- Los métodos y variables *private* no pueden verse por otra clase a excepción de aquella en la que están definidos:

```
public class AprivateClass {

private int aPrivateInt;

private String aPrivateString;

private float aPrivateMethod() {

. . .

}

}
```

- Mantener un freno al acceso estricto en la representación interna de una clase es importante, ya que permite separar el diseño de la implantación, minimizar la cantidad de información que una clase debe conocer sobre otra para hacer su trabajo y reducir la cantidad de cambios de código que se necesita cuando su representación cambia.

CONVENCIONES PARA EL ACCESO A LAS VARIABLES DE INSTANCIA

- Una buena regla es que, a menos que una variable de instancia sea constante, deberá ser casi siempre private.
- Si no se hace esto, se pueden tener problemas como el siguiente:

```
public class AfoolishClass {

public String aUsefulString;

..... // se setea el valor útil del string

}
```

- Esta clase puede haber sido pensada para configurar a aUsefulString para usarla en otras clases, y esperar que éstas sólo la lean. Puesto que no es privada, estas clases pueden ser:

```
AfoolishClass aFC = new AfoolishClass();

aFC.aUsefulString = oops!;
```

- Como no hay manera de especificar por separado el nivel de protección para leer y escribir en las variables de instancia, casi siempre deberán ser private.

METODOS DE ACCESO

- Si las variables de instancia son private, deben tener acceso al mundo exterior. Esto se lo hace mediante métodos de acceso:

```
public class AcorrectClass {

private String aUsefulString;

public String aUsefulString() { // regresar (get) el valor

return aUsefulString;

}

private protected void aUsefulString(String s) {// establecer (set) el valor

aUsefulString = s;

}

}
```

- Hacer este tipo de implementación permite obtener programas mejores y reutilizables.
- Al separar la lectura y la escritura de la variable de instancia, se puede especificar un método public para que regrese su valor y un método protected para establecerlo.
- Al hacer esto, otras clases de objetos pueden pedir ese valor, pero sólo el programador y sus subclases podrán cambiarlo.
- Si es una parte privada de información, podría hacerse private al método establecer (set) y protected al método obtener (get), o cualquier otra combinación que permita la sensibilidad de la información a la luz del mundo exterior.
- De acuerdo con el compilador y el lenguaje Java 1.0, es legal tener una variable de instancia y un método con el mismo nombre. Sin embargo, algunas personas podrían confundirse con esta convención.
- Para evitar esto, es útil escribir el nombre del método con los prefijos get y set, como por ejemplo setNombreAlumno().
- Si se hace:

aUsefulString(aUsefulString() + algún texto adicional);

el propio programador estaría alterando el valor de la variable aUsefulString. Por eso es importante protegerse de saber demasiado sobre la propia representación, a excepción de algunos pocos lugares en donde se necesitan conocer esos valores.

- Luego, si se necesita cambiar algo sobre aUsefulString, no se afectará a cada uso de la variable en la clase, como lo haría sin los métodos de acceso. En su lugar, sólo se afectarán las implantaciones de sus métodos de acceso.
- Uno de los poderosos efectos de mantener este nivel de desviación para tener acceso a las variables de instancia es que si, en alguna fecha posterior algún código especial necesita ejecutarse cada vez que se tiene acceso a una variable, como aUsefulString, se puede colocar ese código en un lugar y todos los demás métodos en la clase (y en las de otras clases) llamarán en forma correcta a ese código especial.
- Ejemplo:

```
private protected void aUsefulString(String s) { // el método set

aUsefulString = s;

algunaAccionSobreLaVariable(s);

}
```

- Entonces, se deberán hacer llamadas como la siguiente:

```
x(12 + 5 * x());
```

en vez de

```
x = 12 + 5 * x;
```

pero esto permitirá a futuro tener reutilización y fácil mantenimiento.

VARIABLES DE CLASE Y METODOS

- Son variables que tienen una sola copia y todas las instancias de la clase la compartirán.

```
public class Circulo {

public static float pi = 3.1415925F;

public float area(float r) {

return pi * r * r;

}

}
```

- Las instancias se pueden referir a sus propias variables de clase como si fueran variables de instancia, como en el ejemplo anterior.
- Ya que es public, los métodos de las demás clases también pueden referirse a pi:

```
float circunferencia = 2 * Circulo.pi * r;
```

- Los métodos de clase se definen en forma análoga. Pueden tener acceso a ellos en las mismas dos formas que las instancias de sus clases, pero sólo por medio del nombre de la clase completo por instancias de otras clases.
- Ejemplo: definición de una clase que define los métodos de clase para ayudarle a contar sus propias instancias, que se encuentra en el listado InstanceCounterTester.java

El modificador final

- Se lo utiliza de varias maneras:
- Cuando se lo aplica a una clase, significa que la clase no puede subclasificarse.
- Si se aplica a una variable, indica que la variable es constante.
- Si se aplica a un método, indica que el método no puede ser sobrepuesto por las subclases.

Clases final

- Formato:

```
public final class AfinalClass {

. . .

}
```

- Se debe declarar una clase final sólo por dos causas:
- Por seguridad, ya que se espera usar sus instancias con capacidades no modificables, por lo que se requerirá que ninguna otra persona sea capaz de subclasificar y crear nuevas y diferentes instancias de ellas.
- Por eficiencia, porque se desea contar con instancias sólo de esa clase (y no subclases), de modo que se las pueda optimizar.

Variables final

- Se utilizan para declarar constantes en Java, de la siguiente manera:

```
public class AnotherFinalClass {

public static final int aConstantInt = 123;

public final String aConstatString = Hello world!);

}
```

- El espacio en la última línea hace más comprensible que la primera variable es de clase y la de abajo no, pero que las dos son public y final.
- Las variables de instancia y de clase final pueden utilizarse en expresiones al igual que las variables de clase y de instancia, pero no pueden modificarse.
- Por esta razón, a las variables final deben dárseles sus valores (constantes) en el momento de la declaración.
- Las clases ofrecen constantes útiles a otras por medio de las variables de clase final como aConstantInt (constante entera) en el anterior ejemplo.
- Otras clases las referencian como antes: AnotherFinalClass.aConstantInt.
- Las variables locales no pueden declararse como final.
- Las variables locales no pueden tener modificadores frente a ellas:

```
{

Int aLocalVariable;

. . .

}
```

Métodos final

- Ejemplo:

```
public class MyPenultimateFinalClass {

public static final void aUniqueAndReallyUsefulMethod() {

. . .

}

public final void noOneGetsToDoThisButMe() {

. . .

}

}
```

- Los métodos final no pueden ser sobrepuestos por las subclases.
- El uso de final para métodos es por eficencia.
- Haciéndolo así, el compilador puede diferenciar al método de entre los demás, ya que sabe que nadie más puede subclasificar y sobreponer el método para cambiar su significado.

- La biblioteca de clase Java declara muchos de los métodos final más usados para beneficiarse con el aumento de velocidad.
- En el caso de las clases final, esto tiene mucho sentido y es una elección adecuada.
- Los métodos private son en efecto final, como lo son todos los métodos declarados en una clase final.
- Si se utilizan demasiado los métodos de acceso y la preocupación por la eficiencia es exagerada, veamos una nueva versión de AcorrectClass que es mucho más rápida:

```
public class AcorrectFinalClass {

private String aUsefulString;

public final String aUsefulString() {

return aUsefulString;

}

private protected final void aUsefulString(String s) {

aUsefulString = s;

}

}
```

Métodos y clases abstract

- Cuando se crea una superclase con el fin de que actúe sólo como un depósito común y compartido, y si sólo sus subclases esperan ser utilizadas (instanciadas), entonces a esa superclase se la llama una clase abstract (abstracta).
- Estas clases no pueden crear instancias, aunque contienen lo que una clase normal posee y además, pueden prefijar algunos de sus métodos con el modificador abstract.
- A las clases no abstractas se les impide utilizar este modificador; usarlo incluso en uno de sus métodos requerirá que su clase completa se declare como abstract.
- Ejemplo:

```
public abstract class MyFirstAbstractClass {

int anInstanceVariable;

public abstract int aMethodMyNonAbstractSubclassesMustImplement();

public void doSomething {

. . . // método normal

}

}

public class AconcreteSubClass extends MyFirstAbstractClass {
```

public int aMethodMyNonAbstractSubclassesMustImplement() {

. . . // implementación obligatoria del método

}

}

- Los métodos abstract no necesitan implementación, pero se requiere que las subclases abstractas tengan una implementación.
- La clase abstract sólo ofrece la plantilla para los métodos que serán implementados por otros más adelante.

47

Objeto punto

pt1

x: 200

y: 200

pt2