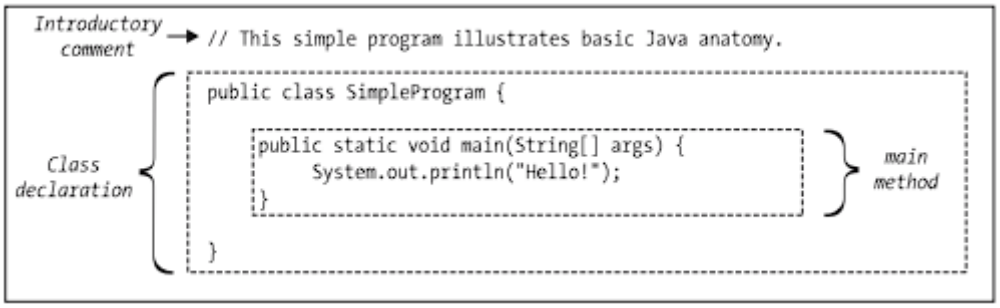
Anatomía simple de un programa de Java
La estructura de un programa de java es:

Comentario introductorio donde se explique que hace el programa



Declaracion de clases **El metodo principal**

En el comentario introductorio se pueden usar 3 diferentes estilos:

Comentario tradicional

este tipo de comentario comienza con “/*” y termina con “*/”

asi se puede usar

/* aqui va el comentario en lenguaje normal diciendo que hace y que cosas se deberian tener en cuenta para el correcto funcionamiento */

Comentarios en linea final

este tipo de comentario comienza con “//” como aveces lo haciamos en bases en los sql para escribir cosas que no se ejecutaban por ejemplo:

// lo que va aqui no se tendra en cuenta en el codigo de ejecucion.
// y lo siguiente menos
// y asi sucesivamente

Comentarios de documentacion en Java

Estos son conocidos como comentarios Javadoc, que se generan mediante una utilidad en linea de comandos, pero no se habla al respecto hasta el capitulo 13.

Declaracion de las clases

Con estructura

public class Nombre de la clase {

…

}

Donde **“...”** significa el cuerpo de la clase donde esta la logica del programa

El metodo principal

Sirve como punto de entrada a el programa con estructura

public static void main(String [] args) {

}

y dentro va la logica del programa como por ejemplo

System.out.println(“Hello!”);

lo cual imprimira “Hello!”

Las mecanicas de Java

Para compilar y ejecutar archivos .java debemos hacerlo en el cmd de nuestros dispositivos, asi cuando estamos en el directorio donde estan los archivos que queremos, ejecutamos

javac nombredelarchivo.java

Si hay varios archivos .java que queremos que se ejecuten todos a la vez, podemos hacer

javac *.java

Si la ejecucion resulta exitosa, entonces aparecera un nuevo archivo en nuestro directorio el cual se llamara “nombre de ejecucion inicial.class” este resultado lo llaman **Bytecode**.

Ahora para ejecutar ese **Bytecode** debemos escribir en otro comando.

java bytecode_filename pero esta vez sin escribir el .class con el que se genero

Hay que hacerlo asi, ya que si no , entonces debemos informar a la ejecucion de donde estamos queriendo sacar los documentos o el documento .class que buscamos para la ejecucion de esta manera.

For example, on DOS/Windows:

```
java -cp C:\home\javastuff;D:\reference\workingdir;S:\foo\bar\files SimpleProgram
```

or on Solaris/Linux:

```
java -cp /barkerj/work:/java/examples/ex1 SimpleProgram
```

Tipos primitivos

Cuando una variable es definida a su ves debe ser definido su tipo, en java este tipo debe contener infomacion sobre cuanta memoria debe ser almacenada en la variable cuando se ejecute, asi se definen 8 tipos primitivos que hablan sobre esto.

4 tipos para datos numericos enteros

byte: 8-bit
short: 16-bit
int: 32-bit
long: 64-bit

2 tipos para datos punto flotante

float: 32-bit
double: 64-bit

2 tipos adicionales

char: de un solo caracter usa 16-bit
boolean: que solo asume true o false, los cuales son mayormente usados como banderas para activar funciones.

2.5 VARIABLES

Antes de que se pueda usar una variable en un programa Java, el tipo y el nombre de la variable deben declararse en el compilador de Java, por ejemplo: `int count;`
La asignación de un valor a una variable se logra utilizando el operador de asignación de Java, =. Una declaración de asignación consiste en un nombre de variable previamente declarado a la izquierda de la =, y una expresión que se evalúa al tipo apropiado a la derecha de la =. Por ejemplo: `int count = 1;`

Se puede proporcionar un valor inicial cuando se declara una variable por primera vez. Una variable se puede declarar en una sentencia, y luego se le puede asignar un valor en una declaración separada más adelante en el programa. Además, se puede asignar

un valor a una variable booleana usando los literales *true* o *false* y también se puede asignar un valor literal a una variable de tipo char encerrando el valor (un solo carácter Unicode) entre comillas simples.

El uso de comillas dobles ("...") está reservado para asignar valores literales a las variables de cadena. Lo siguiente no se compilaría en Java: `char c = "A";`

Convenciones de nomenclatura variable

Los nombres de variables válidos en Java deben comenzar con un carácter alfabético, una puntuación inferior o un signo de dólar y pueden contener cualquiera de estos caracteres más dígitos numéricos. No se permiten otros caracteres en los nombres de variables.

The following are all valid variable names in Java:

```
int simple;               // starts with alphabetic character
int _under;               // starts with underscore
int more$money_is_2much;  // may contain dollar signs, and/or underscores, and/or
                          // digits, and/or alphabetic characters
```

while these are invalid:

```
int 1bad;                 // inappropriate starting character
int number#sign;          // contains invalid character
int foo-bar;              // ditto
int plus+sign;            // ditto
int x@y;                  // ditto
int dot.notation;         // ditto
```

La convención que se observa en toda la comunidad de programación de OO es formar nombres de variables utilizando principalmente caracteres alfabéticos, evitando el uso de guiones bajos y, además, adherirse a un estilo conocido como ***camel casing***. Con ***camel casing***, la primera letra de un nombre de variable está en minúsculas, la primera letra de cada palabra concatenada posterior en el nombre de la variable está en mayúsculas y el resto de los caracteres están en minúsculas. Las palabras clave de Java no se pueden usar como nombres de variables. De hecho, el compilador generaría los siguientes dos mensajes de error:

```
not a statement
int public;
^

';' expected
int public;
    ^
```

2.6 VARIABLE INITIALIZATION (Inicialización variable)

En Java, a las variables no se les asigna necesariamente un valor inicial cuando se declaran, pero a todas las variables se les debe asignar un valor antes de que el valor de la variable se utilice en un estado de asignación.

```
int foo;
int bar;
// We're explicitly initializing foo, but not bar.
foo = 3;
foo = foo + bar;   // This line won't compile.
```

The following compiler error would arise on the last line of code:

```
variable bar might not have been initialized
foo = foo + bar;
            ^
```

2.7 THE STRING TYPE (El tipo de cadena)

Una cadena representa una secuencia de cero o más caracteres Unicode.

El símbolo String comienza con una "S" mayúscula, mientras que los nombres de los tipos primitivos se expresan en minúsculas: int, float, boolean, etc. Esta diferencia de mayúsculas es deliberada y obligatoria: ***s*** (minúscula) no funcionará como un tipo:

Hay varias formas de crear e inicializar una variable de cadena. El más fácil y común es declarar una variable de tipo String y asignar a la variable un valor usando un literal de cadena. Una cadena literal es cualquier texto entre comillas dobles, incluso si consiste en un solo carácter.

Dos enfoques de uso común para inicializar una variable de cadena son los siguientes:

• Asignación de una cadena vacía, representada por dos comillas dobles consecutivas.
• Asignando el valor ***null***, que es la palabra clave de Java que se utiliza para señalar que la cadena aún no se ha asignado un valor "real".

El operador del signo más (+) se utiliza normalmente para la suma aritmética, pero cuando se utiliza en la unión con Strings, representa la concatenación de cadenas. Cualquier número de valores de cadena se puede concatenar con el operador +.

2.8 CASE SENSITIVITY (Sensibilidad de la caja)

Java es un lenguaje que distingue entre mayúsculas y minúsculas. Es decir, el uso de mayúsculas frente a minúsculas en Java es deliberado y obligatorio, por ejemplo:

• Los nombres de las variables que se escriben de la misma manera, pero que difieren en el uso del caso, representan diferentes variables:

• Todas las palabras claves están en minúscula: ***public, class, int, boolean*** y ***forth.*** No hay que escribirlas en mayúsculas, ya que el compilador se oponerá violentamente, a menudo con mensajes de error de compilación ininteligibles. El nombre del método principal debe estar en minúsculas.

2.9 JAVA EXPRESSIONS (Expresiones de Java)

Java es un lenguaje orientado a la expresión. Una expresión simple en Java es

• Aconstante: *7, false*

• Un *char*(acter) en comillas sencillas: *'A','3'*

• Una string en comillas dobles: *"foo","Java"*

• Los nombres de cualquier variable declarada: *myString, x*

• Cualquiera de los dos tipos de expresión anteriores que se combinaron con uno de los operadores binarios de Java: *x + 2*

• Cualquiera de los tipos de expresión anteriores que fue codificado por uno de los operadores de Java: *i++*

• Cualquiera de los tipos de expresión anteriores entre paréntesis: *(x + 2)*

Las expresiones de complejidad arbitraria se pueden ensamblar a partir de los diferentes tipos de expresiones simples mediante la anidación de paréntesis.

Operadores aritméticos

El lenguaje Java proporciona una serie de operadores aritméticos básicos.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder (the remainder when the operand to the left of the % operator is divided by the operand to the right; e.g., 10 % 3 = 1, because 3 goes into 10 three times, leaving a remainder of 1)

Los operadores *+* y – se suelen usar para representar números negativos y positivos.
Además del operador de asignación simple, =, hay una serie de operadores de asignación de compuestos especializados, que combinan la asignación de variables con una

operación aritmética.

Operator	Description
+=	a += b is equivalent to a = a + b.
-=	a -= b is equivalent to a = a - b.
*=	a *= b is equivalent to a = a * b.
/=	a /= b is equivalent to a = a / b.
%=	a %= b is equivalent to a = a % b.

Los operadores aritméticos de incremento unario (++) y decremento (--), se utilizan para aumentar o disminuir el valor de una variable ***int*** en 1 o de un valor de ***coma flotante*** (flotante, doble) en 1,0. Se les conoce como operadores unarios porque se aplican a una sola variable, mientras que los operadores binarios combinan los valores de dos expresiones. También se pueden aplicar a las variables de caracteres para avanzar o retroceder una posición de carácter en la secuencia de clasificación Unicode.

Los operadores de incremento y disminución se pueden utilizar de forma de prefijo o postfijo. Si el operador se coloca antes de la variable en la que está operando (modo de prefijo), el incremento o decremento de esa variable se realiza antes de que se utilice el valor actualizado de la variable en cualquier asignación realizada a través de esa declaración.

Por otro lado, si el operador de incremento/decremento se coloca después de la variable en la que está operando (modo postfijo), el incremento o decremento se produce después de que se utilice el valor original de la variable en cualquier asignación realizada a través de esa declaración. Los operadores de incremento y decremento se utilizan comúnmente junto con los bucles.

Operadores relacionales y lógicos

Una expresión lógica compara dos expresiones (simples o complejas) ***exp1*** y ***exp2*** de una manera especificada, resolviendo a un valor booleano de verdadero o falso.
Para crear expresiones lógicas, Java proporciona los operadores relacionales.

Operator	Description
exp1 == *exp2*	`true` if *exp1* equals *exp2* (note use of a ***double*** equal sign for testing equality).
exp1 > *exp2*	`true` if *exp1* is greater than *exp2*.
exp1 >= *exp2*	`true` if *exp1* is greater or equal to *exp2*.
exp1 < *exp2*	`true` if *exp1* is less than *exp2*.
exp1 <= *exp2*	`true` if *exp1* is less than or equal to *exp2*.
exp1 != *exp2*	`true` if *exp1* is not equal to *exp2* (`!` is read as "not").
!exp	`true` if *exp* is false, and `false` if *exp* is true.

Además de los operadores relacionales, Java proporciona operadores lógicos que se pueden utilizar para combinar/modificar expresiones lógicas.

Operator	Description
exp1 && *exp2*	Logical "and"; compound expression is true only if ***both*** *exp1* ***and*** *exp2* are true
exp1 || *exp2*	Logical "or"; compound expression is true if ***either*** *exp1* ***or*** *exp2* is true
!exp	Logical "not"; toggles the value of a logical expression from `true` to `false` and vice versa

Las expresiones lógicas se utilizan más comúnmente con estructuras de control de flujo.

Evaluación de expresiones y precedencia del operador

Las expresiones de complejidad arbitraria se pueden construir colocando entre paréntesis anidados en capas, por ejemplo, (((8 * (y + z)) + y) * x). El compilador generalmente evalúa tales expresiones desde el más interno hasta el más externo, de izquierda a derecha. Suponiendo que x, y y z se declaren e inicialicen.

Luego la expresión en el lado derecho de la siguiente declaración de asignación. Se evaluaría pieza por pieza de la siguiente manera:

En ausencia de paréntesis, ciertos operadores tienen prioridad sobre otros en términos de cuándo se aplicarán en la evaluación de una expresión. Por ejemplo, la multiplicación o división se realiza antes de la suma o la resta. La precedencia del operador se puede alterar explícitamente mediante el uso de paréntesis; las operaciones realizadas dentro de los paréntesis tienen prioridad sobre las operaciones fuera de los paréntesis.

El tipo de expresión

El tipo de expresión es el tipo Java del valor al que la expresión se evalúa en última instancia.

Tipos de conversión automáticos y castings explícitos

Java permite darles a diferentes variables unas conversiones de tipo utilizando las sintaxis preestablecidas, denotando así:

```
int x;
double y;
 y = 2.7;
 x = y;
```

Esto en otro lenguaje daría un aproximado de 2 en entero a la variable x, pero Java directamente mostrara un error de denotación de variables, para poder hacer esto se debe usar la sintaxis:

```
int x;
```

```
double y;

y = 2.7;

 x = (int) y;
```

de esta forma la variable x cumplirá con la condición de ser el valor 2 entero, a esto lo llama conversión de estrechamiento

pero si se intentara hacer de forma contraria; este no tendría problemas:

```
int x;

double y;

x = 2;

y = x;
```

a esto se le puede llamar conversión de ampliación.

El libro da la recomendación de utilizar para los punto flotantes el condicional *double*, puesto que al utilizar *float* sin antes especificar el tipo puede generar un error:

```
float y = 3.5;  //Este no compila por el punto flotante, lo considera conversión estrechamiento

float y = (float) 3.5;  //Este si compila al especificar que es un flotante

float y = 3.5F; //Otra forma de especificar el que es flotante
```

(Cabe recalcar que el libro indica que estos problemas no suceden si directamente colocan el tipo como *double*)

En cambio, si es un valor tipo *char* y se cambia a cualquier tipo numérico, este pasara a ser el numero ASCII equivalente:

```
char c = 'a';

// Assigning a char value to a numeric variable transfers its
// ASCII numeric equivalent value.
int x = c;
float y = c;
double z = c;

System.out.println(x);
System.out.println(y);
System.out.println(z);
```

Here's the output:

```
97
97.0
97.0
```

La única variable que no tiene conversiones implícitas o explicitas son de tipo *boolean*.

Loops y otras formas de control de flujo:

-Clausula IF:

Java da la capacidad de ejecutar bloques de código dependiendo de los condicionales propuestos; para esto los condicionales IF/ELSE ayudan a no correr todo el bloque de código sino solo las partes que cumplan la condición.

Esta tiene sintaxis:

```
// Pseudocode.
if (logical-expression) {
    execute whatever code is contained within these braces
    if logical-expression evaluates to true
}
```

Or, adding an optional `else` clause:

```
// Pseudocode.
if (logical-expression) {
    execute whatever code is contained within these braces
    if logical-expression evaluates to true
}
else {
    execute whatever code is contained within these braces if
    logical-expression evaluates to false
}
```

```
if (x > 3) {
    y = x;
}
else {
    z = x;
}
```

(pequeño ejemplo que da el libro de como poder usarlo)

Se utiliza “==” para hacer una equivalencia entre 2 expresiones.

Se utiliza “!” para generar la negación de la expresión.

Los condicionales también se activan a partir de una sola expresión donde se considera que sea TRUE/FALSE el resultado de esa expresión:

```
if (comprobado) { // equivalente a: if (comprobado == true) {
System.out.println("estado terminado");
 }
if (!comprobado) { // equivalente a: if (comprobado == false)
System.out.println("estado no terminado");
```

```
}
```

además, se puede anidar los condicionales dentro de otros condicionales para lograr abarcar mas casos, pero el libro sugiere que no halla tanta profundidad en los niveles de anidamiento, a cambio nos muestra estas 2 formas de poder escribir condicionales anidados; ambos están correctamente escritos y ya depende del código cual de las 2 se debería usar:

```
if (logical-expression-1) {
    execute this code
}
else {
    if (logical-expression-2) {
        execute this alternate code
    }
    else {
        execute this code if neither of the above expressions evaluate to true
    }
}
```

```
if (logical-expression-1) {
    execute this code
}
else if (logical-expression-2) {
    execute this alternate code
}
else {
    execute this code if neither of the above expressions evaluate to true
}
```

Note that the two forms are logically equivalent.

-Clausula Switch:

Tiene una similitud con la sentencia IF, con la diferencia que no usa comprobantes lógicos sino valores tipo *char o int*, utilizando *cases* diferentes para generar los “condicionales” para los que se requiera usar estas variables:

```
switch (int-or-char-expression) {
    case value1:
        one or more lines of code to execute if value of expression matches value1
        break;
    case value2:
        one or more lines of code to execute if value of expression matches value2
        break;
    // more case labels, as needed ...
     case valueN:
         one or more lines of code to execute if value of expression matches valueN
         break;
     default:
         default code to execute if none of the cases match
}
```

For example:

```
// x is assumed to have been previously declared as an int.
switch (x) {
    case 1:  // executed if x equals 1
      System.out.println("One ...");
      break;
    case 2:  // executed if x equals 2
      System.out.println("Two ...");
      break;
    default:  // executed if x has a value other than 1 or 2
      System.out.println("Neither one nor two ...");
}
```

(por si acaso en el ejemplo “case 1” se refiere a si x es 1 entonces…)

Ademas de varias recomendaciones que hace el libro respecto a la clausula switch (pag 82 del pdf), la mas importante a recalcar es la finalizacion *break*; a diferencia de IF/ELSE, cuando entra a un case, este continua leyendo los demas si es que no está esta etiqueta, por lo que al ponerla deja de leer el bloque de codigo, pero si no se pone seguira leyendo los otros casos (esto puede usarse a favor por si se necesita abarcar mas de 1 caso).

```
switch (x) {
    case 1:
        code to be executed if x equals 1
    case 2:
        code to be executed if x equals 1 OR 2
    case 3:
        code to be executed if x equals 1, 2, OR 3
        break;
    case 4:
        code to be executed if x equals 4
}
```

-Clausula for:

La clausula for, al igual que en python es un ciclo, coon la diferencia que este necesita 3 parametros para comenzar el ciclo: inicializador, condicion, iterador

Se utiliza “++” para ir sumando de a 1.

```
for (int i = 0; condition; iterator) {
    code to execute while condition evaluates to true
}
for (int i = 0; i < 5; i++) {
    code to execute as long as the value of i remains less than 5
}
```

El libro muestra el siguiente ejemplo de 2 ciclos for anidados para hacer una tabla de multiplicar:

```
public class ForDemo {
    public static void main(String[] args) {
        // Compute a simple multiplication table.
        for (int j = 1; j <= 4; j++) {
            for (int k = 1; k <= 4; k++) {
                System.out.println(j + " * " + k + " = " + (j * k));
            }
        }
    }
}
```

Here's the output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
```

-Clausula While:

El ciclo while (tiene el mismo uso que python).

Se utiliza “| |” para el OR.

La sintaxis que se va a utilizar para el ciclo while es:

```
while (condition) {
    code to repeatedly execute while condition continues to evaluate to true
}
```

```
int x = 1;
int y = 1;

while ((x < 20) || (y < 10)) {
    hopefully we'll do something within this loop body that increments the value of
    either x or y, to avoid an infinite loop!
}
```

Jump Statements:

Por último, habla de 2 sentencias que ayudan a la continuación o rompimiento brusco del código, que son: *break y continue.*

La sentencia break se usa para hacer un corte en la ejecución del código de forma brusca (es una mala practica abusar de esta opción) como ya se pudo mostrar en la parte "switch".

Por otro lado, continue ayuda a continuar la ejecución del programa o buble que esta actualmente en proceso, pero omitiéndose el resto de las líneas que están a continuación de esta sentencia; cabe recalcar que no rompe el ciclo solo omite los pasos restantes.

Ejemplo que muestra el libro de break y continue:

```
for (int x = 1; x <= 4; x++) {
    // ... but when x reaches the value 3, we prematurely terminate this
    // loop with a break statement.
    if (x == 3) break;
    System.out.println(x);
}

System.out.println("Loop finished");
```

The output produced by this code would be as follows:

```
1
2
Loop finished
```

```
for (int x = 1; x <= 4; x++) {
    // ... but when x reaches the value 3, we prematurely terminate
    // this iteration of the loop (only) with a continue statement.

    if (x == 3) continue;
    System.out.println(x);
}

System.out.println("Loop finished");
```

The output produced by this code would be as follows:

```
1
2
4
Loop finished
```

Lenguajes estructurados en bloques y el alcance de una variable

java es un lenguaje estructurado en bloques.

Bloque de código: es una serie de cero o más líneas de código encerradas en llaves {...}. los bloques pueden estar anidados dentro de otros en cualquier grado de profundidad

variables

la variables pueden ser declaradas en cualquier bloque del programa

alcance de una variable: es una porción de código en la que la variable puede ser referenciada por su nombre

una variable esta dentro del alcance si el compilador la reconoce por su nombre. una vez se ejecute el bloque de código, cualquier variable declarada dentro estará fuera del alcance y será inaccesible para el programa

Una variable puede ser accesible a cualquier bloque interior de código que es posterior a su declaración

imprimiendo en la pantalla

para imprimir un mensaje en la pantalla se usa la siguiente línea

System.out.println(expresión a imprimir);

print vs println

println imprime el mensaje con un salto de línea al final mientras que print no.

para separar mensajes muy largos y hacer el código más entendible se pueden separar los string con + dentro del código para concatenarlos en la salida

Secuencias de escape

Secuencia de Escape	Descripción
\n	Salto de línea
\t	Tabulador
\\	Diagonal Inversa \
\"	Comillas Dobles
\'	Comilla Simple
\r	Retorno de Carro (Solo en modo Administrador)
\b	Borrado a la Izquierda (Solo en modo Administrador)

Elementos del estilo Java

Algunas pautas y convenciones

uso apropiado de la identación

Las declaraciones dentro de un bloque de código deben estar identadas según la línea inicial/final del bloque al que pertenecen

a veces se tienen muchos niveles de indentación o declaraciones muy largas. Para evitar esto, es mejor dividir la línea con espacios en blanco o límites de puntuación, indentando continuación con respecto al inicio de la línea

Comentarios

- Si puede haber alguna duda sobre lo que hace un pasaje de código, haz un comentario
- Indenta al mismo nivel que el bloque de código o declaración al que pertenece. aplica.
- Asegúrese de que todos los comentarios agreguen valor; no diga lo obvio

Colocación de los {}
se puede hacer de dos maneras

- colocando una llave de apertura al final de la línea de código que inicia un bloque determinado. Cada llave de cierre va en su propia línea, alineada con el primer carácter de la línea que contiene la llave de apertura:

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
        }
    }
}
```

- Cada llave de cierre va en su propia línea como antes, alineada con la llave de apertura correspondiente que también irá en su propia línea

```
public class Test
{
    public static void main(String[] args)
    {
        for (int i = 0; i < 3; i++)
        {
            System.out.println(i);
        }
    }
}
```

Nombres de variables

El objetivo al elegir nombres de variables es hacer un programa lo más legible y auto documentable posible. Evite el uso de letras individuales como nombres de variables, excepto para las variables de control de bucle.

Escriba toda la palabra como variable de ser posible, si se abrevia asegúrese de que seguirá siendo entendida.

3.1 INTRODUCCIÓN Y SOFTWARE AT ITS SIMPLEST - 3.2 WHAT IS AN OBJECT?

Esta sección hace hincapié en que todo software consta de datos y funciones que operan con esos datos. Por ejemplo, se menciona que un software puede constar de datos como el nombre de un estudiante y funciones que operan con esos datos, como calcular su nota media.

Descomposición funcional frente al enfoque orientado a objetos

Se compara el enfoque de descomposición funcional con el enfoque orientado a objetos, resaltando las deficiencias del primero y cómo el segundo las remedia. Por ejemplo, se menciona que, con la descomposición funcional, los datos son pasados de una función a otra, lo que puede causar efectos secundarios significativos si la estructura de datos cambia. En contraste, con el enfoque orientado a objetos, cada objeto es responsable de la integridad de sus propios datos, lo que evita efectos secundarios globales.

El enfoque orientado a objetos
Se destaca cómo el enfoque orientado a objetos aborda las deficiencias del enfoque tradicional, incluyendo la encapsulación de datos y la responsabilidad de los objetos en la integridad de sus datos. Por ejemplo, se menciona que, en el enfoque orientado a objetos, cada objeto es responsable de garantizar la integridad de sus propios datos, lo que simplifica la detección y corrección de errores.

Objetos y clases
Se explican las ventajas de la aproximación orientada a objetos, cómo usar clases para especificar datos y comportamientos de un objeto, y la creación de objetos en tiempo de ejecución. Por ejemplo, se menciona que las clases pueden definir las características comunes de un grupo de objetos, como la clase "curso" que define los datos y comportamientos compartidos por todos los cursos.

¿Qué es un objeto?
Se define un objeto como una construcción de software que combina estado (datos) y comportamiento (funciones) para representar una abstracción de un objeto del mundo real. Por ejemplo, se puede considerar un objeto "estudiante" que tiene datos como nombre, número de identificación y promedio, junto con comportamientos como inscribirse en un curso o verificar requisitos.

Figure 3-1. *At its simplest, software consists of data and functions that operate on that data.*

Estado/Datos/Atributos
Se detalla los datos que un objeto puede requerir y cómo estos representan el estado del objeto. Por ejemplo, se mencionan atributos como el nombre del estudiante, su número de identificación, su promedio académico y los cursos en los que está inscrito.

Comportamientos/Operaciones/Métodos
Se describe las operaciones que un objeto puede realizar para acceder y modificar sus atributos. Por ejemplo, se mencionan operaciones como inscribirse en un curso, verificar requisitos y obtener el horario de clases.

3.3 WHAT IS A CLASS? – 3.6 USER-DEFINED TYPES AND REFERENCE VARIABLES
Un "clase" es una abstracción que describe las características comunes de todos los objetos en un grupo de objetos similares. Por ejemplo, se podría crear una clase llamada

"Estudiante" para describir todos los objetos de estudiante reconocidos por el SRS. Una clase define la estructura de datos (es decir, los nombres y tipos de atributos) de cada objeto perteneciente a esa clase, así como las operaciones o métodos que pueden ser realizados por esos objetos y cómo se invocan formalmente esas operaciones. Además, especifica qué acciones debe tomar un objeto detrás de escena para llevar a cabo esas operaciones.

Los lenguajes de programación orientados a objetos, incluyendo Java, siguen las siguientes convenciones de nombres:
- Al nombrar clases, comenzamos con una letra mayúscula, pero usamos mayúsculas y minúsculas para el nombre en general. Por ejemplo: Estudiante, Curso, Profesor, etc. Si el nombre de una clase idealmente sería una frase compuesta por varias palabras, como "catálogo de cursos", comenzamos cada palabra con una letra mayúscula y las concatenamos sin usar espacios, guiones o guiones bajos para separarlas, por ejemplo: CatalogoDeCursos. Este estilo se conoce como Pascal casing.
- La convención para los nombres de atributos y métodos es comenzar con una letra minúscula, pero capitalizar la primera letra de cualquier palabra subsiguiente en el nombre. Por ejemplo, los nombres de atributos típicos podrían ser nombre, idEstudiante, o cargaCurso, mientras que los nombres de métodos típicos podrían ser inscribirEnCurso e imprimirTranscript. Este estilo se conoce como camel casing.

Una vez que hemos determinado la estructura de datos común y los comportamientos que deseamos impartir a un conjunto de objetos similares, debemos declararlos formalmente como atributos y métodos en el contexto de una clase Java. Esta declaración de clase residiría en un archivo fuente llamado NombreClase.java (por ejemplo, Student.java) y sería posteriormente compilada en forma de bytecode en un archivo llamado NombreClase.class.

Instancia
La definición de una clase puede ser pensada como una plantilla para crear objetos de software, un "patrón" utilizado para:

- Asignar una cantidad prescrita de memoria dentro del JVM para alojar los atributos de un nuevo objeto.
- Asociar un cierto conjunto de comportamientos con ese objeto.

El término "instanciación" se utiliza para referirse al proceso mediante el cual se crea un objeto en memoria en tiempo de ejecución basado en una definición de clase. A partir de una sola definición de clase, por ejemplo, "Estudiante", podemos crear muchos objetos con estructuras de datos y comportamientos idénticos, de la misma manera que usamos un solo cortador de galletas para hacer muchas galletas todas del mismo tamaño. Otra forma de referirse a un objeto, entonces, es como una instancia de una clase particular, por ejemplo, "Un objeto Estudiante es una instancia de la clase Estudiante". Las clases pueden diferenciarse de los objetos de la siguiente manera:

- Una clase define las características (atributos, métodos) que cada objeto perteneciente a la clase debe poseer; por lo tanto, una clase puede pensarse como una plantilla de objetos.

Encapsulamiento

La encapsulación es un término formal que se refiere al mecanismo que combina el estado y el comportamiento de un objeto en una única unidad lógica. Todo lo que necesitamos saber sobre un estudiante dado está, en teoría, contenido dentro de los límites de un objeto Estudiante, ya sea directamente como un atributo de ese objeto o indirectamente como un método que puede responder una pregunta o tomar una determinación sobre el estado del objeto.

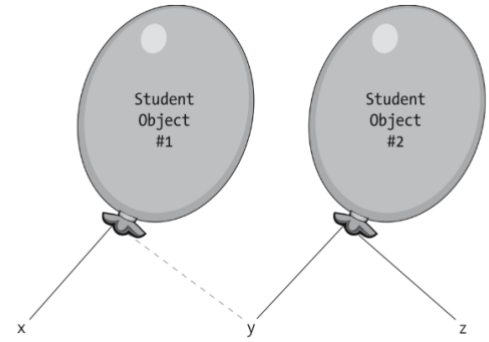
Variables de referencia

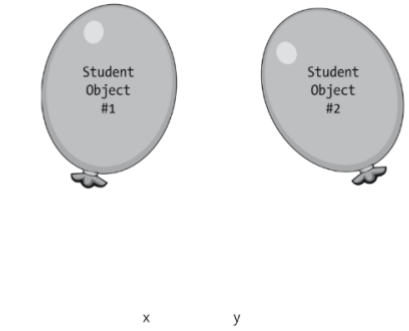
En resumen, en un lenguaje no orientado a objetos como C, la declaración int x; significa que x es un nombre simbólico que hemos creado para referirnos a un valor entero que se almacena en la memoria de la computadora. En contraste, en un lenguaje orientado a objetos como Java, al declarar una variable de tipo Student, por ejemplo Student y;, y es un nombre simbólico que usamos para referirnos a un objeto Student (una instancia de la clase Student) que se almacena en la memoria de la computadora. Esta variable y se conoce como una variable de referencia, ya que se refiere a un objeto. Mientras que en Java, los tipos primitivos como int, double, etc., no son variables de referencia porque no se refieren a objetos

Instanciación de objetos: una mirada más cercana

Varias variables de referencia pueden referirse simultáneamente al mismo objeto. Sin embargo, una variable de referencia sólo puede referirse a un objeto a la vez.

Por lo tanto, si una variable de referencia ya está haciendo referencia a un objeto, debe soltar ese objeto para hacer referencia a un objeto diferente. Si llega un momento en que todos los manejadores de un objeto en particular se han soltado, entonces ese objeto ya no es accesible a nuestro programa, como un globo de helio que se ha soltado.

Student x = new Student();
 	
Student y;
 	
y = x;
 	
Student z = new Student();
 	
y = z;
	

x = null;
 	
y = null;
 	
z = null;
	

Recolección de residuos

Resulta que si se liberan todos los manejadores de un objeto, podría parecer que la memoria que el objeto ocupa dentro de la JVM (Máquina Virtual de Java) se desperdiciaría permanentemente.

La JVM realiza periódicamente la recolección de basura, un proceso que recupera automáticamente la memoria de los objetos "perdidos" mientras se ejecuta una aplicación. Así es como funciona el recolector de basura de Java:

- Si no quedan referencias activas a un objeto, éste se convierte en candidato a la recogida de basura.

- Sin embargo, el recolector de basura no recicla inmediatamente el objeto, sino que la recolección de basura se produce cada vez que la JVM determina que la aplicación se está quedando sin memoria libre, o cuando la JVM está inactiva.

- Por lo tanto, durante algún tiempo, el objeto "huérfano" seguirá existiendo en la memoria. Simplemente no tendremos ningún manejador/variable de referencia con el que acceder a él.

Tenga en cuenta que hay una manera de solicitar explícitamente la recolección de basura que se produzca en Java a través de la siguiente declaración:

```
Runtime.getRuntime().gc();
```

3.8 Objects As Attributes - 3.9 Three Distinguishing Features of an Object-Oriented Programming Language

El autor se enfoca en la creación de objetos en Java. El libro presenta una guía detallada sobre cómo crear objetos en Java, desde los conceptos básicos hasta la creación de objetos más complejos. En estas páginas, el autor presenta el concepto de herencia y cómo se puede utilizar para crear nuevas clases que heredan los atributos y métodos de una clase existente. La herencia también se puede utilizar para crear una jerarquía de clases que represente una estructura de objetos más compleja. Además, el autor presenta el concepto de polimorfismo y cómo se puede utilizar para crear una interfaz común para un conjunto de clases relacionadas. El polimorfismo permite que las clases se comporten de manera diferente según el contexto en el que se utilizan.

Se utiliza un caso de estudio de un sistema de registro de estudiantes para llevar al lector desde los conceptos básicos de objetos hasta la creación de código real para una aplicación completa. El libro también cubre una discusión tecnológicamente neutral de los principios de construcción de una arquitectura de tres capas utilizando Java, lo que introduce la noción de separación de capas de modelo - capa de presentación - capa de datos. Los ejemplos de código utilizados en todo el libro son versiones de Java estándar. Es decir que funcionará sin importar la versión de Java que se use.