Construcción de bloques

Java Development Kit (JDK) tiene el software mínimo necesario para desarrollar en Java. Este incluye:

- **javac**: convierte los .java en bytecode .class
- **java**: ejecuta un programa java
- **jar**: empaquetador de archivos
- **javadoc**: generador de documentación

Después de convertir a bytecode la JVM puede ejecutar el código en cualquier dispositivo donde este instalado Java. El JRE es una versión reducida de JDK que permite ejecutar programas en Java pero no compilarlos.

Cada 6 meses sale una versión de java. java 17 salio en setiembre del 2021

Estructura de una clase

Una clase es una estructura de código que agrupa elementos, para utilizarla se debe crear un objeto que es una instancia en memoria de la clase.

una clase java tiene 2 elementos primarios métodos (funciones, procedimientos) y campos (variables). Ambos son llamados miembros de la clase. los campos almacenan el estado del programa y los métodos operan sobre ese estado.

```
1  public class Animal{
2
3   String name;

5   public String getName(){
6    return name;
7   }
8
9   public void setName(String newName){
10    name = newName;
11   }
12  }
```

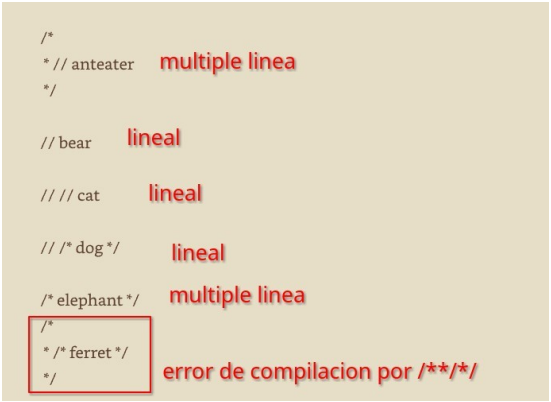
Se declaro una variable, 2 métodos, en el primer método se devuelve un valor y en el segundo se utiliza "void" para indicar que el método no devolverá nada.

Comentarios

los comentarios en Java pueden ser de tres tipos:

```
1  // comentarios de una linea
2
3  /* comentario de
4   *  multiples lineas
5   */
6
7  /**
8   *Javadoc, comentario de documentación
9   *@author Jeanna and Scoot
10  */
```

el iniciar con /** le dice a la herramienta javadoc del JDK que debe procesarlo ya que es un comentario de documentación de la aplicación.



Clase y archivos origen

Lo normal es definir cada clase Java en su propio archivo (la clase es un tipo top-level) pero se pueden definir varias en un mismo archivo. Una clase top-level es a menudo publica.

Cuando ponemos 2 clases en un archivo una debe ser definida como top-level y ser definida como publica ademas de el archivo se debe llamar como esta clase, en este ejemplo el archivo se llamara ***Animal.java***

```
1  public class Animal{
2   private String name;
3  }
4
5  class Animal2{}
```

Escribiendo un método main()

Un programa Java inicia con la ejecución del método main() ya que este es el punto de entrada para que la JVM trabaje.

```
1 public class Zoo{
2  public static void main(String[] args){
3   System.out.println("Hello World");
4  }
5 }
```

- La palabra ***public*** es un modificador de acceso y define el nivel de exposición del método
- La palabra ***static*** enlaza directamente el método con la clase por lo que no se requiere crear una instancia de la clase para llamarlo
- La palabra ***void*** representa el tipo de retorno, se usa void cuando el método no va retornar nada.
- Por ultimo el parámetro definido para le método puede ser nombrado de cualquier forma pero con solo estos 3 tipos de formatos:

```
1 String[] args
2 String args[]
3 String... args
4
5 String[] perros
6 String perros[]
7 String... perros
```

Podemos pasar argumentos al método main y mostrarlos por ejemplo:

```
1 public class Zoo{
2    public static void main(String[] args){
3       System.out.println(args[0]);
4       System.out.println(args[1]);
5    }
6 }
```

al invocar el programa ***javac Zoo diego***
si no pasamos uno de los valores en la ejecución obtendremos una excepción

Entendiendo la importación de paquetes

Java agrupa las clases que tiene construida en paquetes

Para utilizar las clases de estos paquetes hay que señalarlos y para eso se utiliza la sentencia ***import*** que es una instrucción

Los nombres de paquetes son jerárquicos por ende se leen de izquierda a derecha hasta llegar a la clase que se desea utilizar.

existen comodines (**wilcards**) para facilitar la importación de paquetes

import java.util.*; // **importa todas las clases de ese paquete**

la importación con comodín solo permite importar clases que están dentro del paquete, no permite importar sub-paquetes, campos o métodos.

La única clase que nunca es requerido importar es ***java.lang.**** ya que Java siempre lo hará por defecto.

En un ***import*** solo se puede utiliza el comodín al final de la importación.

Si importamos dos paquetes con wilcard y ambos tienen una clase que se llama igual (***java.util.Date*** y ***java.sql.Date***) se producirá un error de compilación

```
1 import java.util.*;
2 import java.sql.*;
```

podemos definir de forma explicita la clase Date que vamos a utilizar para que todo trabaje bien:

```
1 import java.util.Date;
2 import java.sql.*;
```

si ponemos ambas clases de forma explicita, Java nos dará un error de compilación por ambigüedad.

Por defecto las clases que creamos van a estar en el paquete por defecto del proyecto, asumamos estas 2 clases:

```
1  package packagea;
2  public class ClassA {}
3
4  package packageb;
5  import package.ClassA;
6
7  public class ClassB {
8  public static void main(String[] args){
9   ClassA a;
10   System.out.println("Got it");
11 }
12 }
```

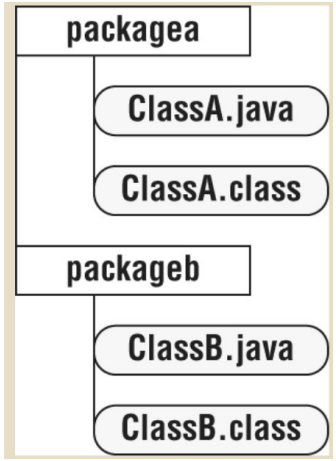


Figura 1: Estructura de los paquetes y clases compilados

Uno puede definir un directorio diferente para almacenar las clases compiladas. Por ejemplo podemos compilar en una carpeta **classes** los archivos anteriores

javac -d classes packagea/ClassA.java packageb/ClassB.java

la ruta de la clase compilada seria **classes/pacageb/ClassB.class**

java -cp classes packageb.ClassB
java -classpath classes packageb.ClassB
java --class-path classes packageb.ClassB

la importancia de estas opciones es que sirven para poder encontrar las clases a ejecutar

si nuestras clases para ejecutarse necesitan paquetes JAR externos podemos agregarlo al class path para que nuestro programa funcione al ejecutarlo.

java -cp ".:/home/files:/tmp/myJar.jar" mypackage/myClass
java -cp ".:/home/files:/tmp/*" mypackage/myClass

podemos crear comprimidos JAR por nuestra cuenta para esto nos ubicamos en el directorio principal de nuestro archivos y ejecutamos el comando:

jar -cvf myNewFile.jar
jar --create--verbose--file myNewFile.jar

podemos indicar un archivo diferente al actual para comprimir

jar -cvf myNewFile.jar -C dir .

dir: es la ruta del directorio que elegimos comprimir
. : el punto significa que el archivo JAR se dejara en la ubicación actual

```
package structure;    // package must be first non-comment
import java.util.*;   // import must come after package
public class Meerkat { // then comes the class
  double weight;     // fields and methods can go in either order
  public double getWeight() {
    return weight; }
  double height;   // another field - they don't need to be together
}
```

Figura 2: orden de archivos en una clase java (Package > Import > Class)

Creando Objetos

Para crear un objeto el cual es una instancia de una clase debemos de utilizar la palabra ***new***.

Park p = new Park();

dentro de la clase se debe definir un método constructor al cual se le llama igual que la clase y que permite definir cosas para cuando se creen objetos.

```
1 public class Chick{
2  public Chick(){
3   System.out.println("in constructor");
4  }
5 }
```

tener en cuenta que el método no debe tener configurado ***VOID*** o un tipo de dato de retorno, sino se vuelve un método mas y no se considera constructor.

La razón del constructor es inicializar variables, otra forma es definir el valor inicial al declarar la variable en la clase. si no se declara ningún constructor en la clase, Java por defecto declara un constructor vació.

```
1 public class Chicken {
2  int numEggs = 12; // inicializado en la misma linea de la declaración
3  String name;
4  public Chicken() {
5   name = "Dake"; // inicializado desde el constructor.
6  }
7 }
```

A continuación muestro un ejemplo de como acceder a las variables después de instanciarlas.

```
1 public class Swan{
2  int numberEggs;
3  public static void main(String[] args) {
4   Swan mother = new Swan();
5   mother.numberEggs = 1;
6   System.out.println(mother.numberEggs);
7  }
8 }
```

How many blocks do you see in the following example? How many instance initializers do you see?

hay 4 bloques de código

```
1: public class Bird {
2:   public static void main(String[] args) {
3:     { System.out.println("Feathers"); }
4:   }
5:   { System.out.println("Snowy"); }  instancia de clase, ya que esta fuera de un metodo
6: }
```

```
1: public class Chick {
2:   private String name = "Fluffy";
3:   { System.out.println("setting field"); } (1)
4:   public Chick() {
5:     name = "Tiny";
6:     System.out.println("setting constructor"); (2)
7:   }
8:   public static void main(String[] args) {
9:     Chick chick = new Chick(); (2)
10:    System.out.println(chick.name); } } (3)

Running this example prints this:

setting field
setting constructor
Tiny
```

Tener en cuenta que no se puede referir "llamar" a una variable que no ha sido inicializada previamente.

Java tiene 2 tipos de datos (***data types***) primitivos (***primitive***) y referidos (***reference***).

Keyword	Type	Min value	Max value	Default value	Example
boolean	true or false	n/a	n/a	false	true
byte	8-bit integral value	-128	127	0	123
short	16-bit integral value	-32,768	32,767	0	123
int	32-bit integral value	-2,147,483,648	2,147,483,647	0	123
long	64-bit integral value	-2^{63}	$2^{63}-1$	0L	123L
float	32-bit floating-point value	n/a	n/a	0.0f	123.45f
double	64-bit floating-point value	n/a	n/a	0.0	123.456
char	16-bit Unicode value	0	65,535	\u0000	'a'

Java tiene ocho tipos de datos integrados, denominados tipos primitivos de Java. Estos ocho tipos de datos representan los elementos básicos de los objetos de Java, ya que todos los objetos de Java son simplemente una colección compleja de estos. Dicho esto, un primitivo no es un objeto en Java ni representa un objeto. Un primitivo es simplemente un valor único en la memoria, como un número o un carácter.

ademas hay que tener en cuenta:

Los tipos ***byte***, ***short***, ***int*** y ***long*** se utilizan para valores enteros sin puntos decimales.

Cada tipo numérico utiliza el doble de bits que el tipo similar más pequeño. Por ejemplo, ***short*** utiliza el doble de bits que ***byte***.

Todos los tipos numéricos tienen signo y reservan uno de sus bits para cubrir un rango negativo. Por ejemplo, en lugar de que byte cubra de 0 a 255 (o incluso de 1 a 256), en realidad cubre de -128 a 127.

Un ***float*** requiere la letra f o F después del número para que Java sepa que es un ***float***. Sin una f o F, Java interpreta un valor decimal como un ***double***

Un ***long*** requiere la letra l o L después del número para que Java sepa que es un ***long***. Sin una l o L, Java interpreta un número sin punto decimal como un ***int*** en la mayoría de los casos.

La principal diferencia entre ***short*** y ***char*** es que ***short*** tiene valores negativos y ***char*** solo positivos desde el 0.

Por defecto Java considera los números definidos como enteros, por ejemplo:

long max = 3213456789; // no compilara ya aunque se defina como ***long*** Java lo ve como entero.

long max = 3213456789L; // de esta forma Java recién lo considerara un ***long***

Java permite expresar números en otros formatos como octal (0 a 7 | 017), hexadecimal (0-9 y a-f | 0xFF) y binario (0-1 | 0B10). el 0x o 0X se pone siempre al inicio de un número hexadecimal y el 0b o 0B se debe poner al inicio de un numero binario.

Java también permite utilizar guiones bajos para nombrar números grandes.

int million1 = 1_000_000;

Los guiones bajos pueden ir juntos pero no es recomendado; algunos ejemplos:

```java
double x1 = _1000.00; // no compila
double x2 = 1000.00_; // no compila
double x3 = 1000_.00; // no compila
double x4 = 1_00_0.0_0; // si compila
double x5 = 1____2; // si compila
```

Un tipo de referencia hace referencia a un objeto (una instancia de una clase). A diferencia de los tipos primitivos que guardan sus valores en la memoria donde se asigna la variable, la referencia "apunta" a un objeto almacenando la dirección de memoria donde se encuentra el objeto, un concepto al que se hace referencia como puntero. A diferencia de otros lenguajes, Java no permite conocer cuál es la dirección de memoria física. Solo se puede utilizar la referencia para hacer referencia al objeto.

String greeting -> la variable ***greeting*** solo puede tener una referencia al objeto String.

Un valor se asigna por referencia de una de dos formas:

- una referencia puede ser asignada a otro objeto del mimos o un tipo compatible.
- una referencia puede ser asignada a un nuevo objeto usando la palabra ***new***

por ejemplo en el siguiente código:

greeting = new String("hola mundo");

greeting es una referencia al nuevo objeto y este objeto solo puede ser accedido vía la referencia que es ***greeting***

TABLE 1.7 Wrapper classes

Primitive type	Wrapper class	Wrapper class inherits Number?	Example of creating
boolean	Boolean	No	Boolean.valueOf(true)
byte	Byte	Yes	Byte.valueOf((byte) 1)
short	Short	Yes	Short.valueOf((short) 1)
int	Integer	Yes	Integer.valueOf(1)
long	Long	Yes	Long.valueOf(1)
float	Float	Yes	Float.valueOf((float) 1.0)
double	Double	Yes	Double.valueOf(1.0)
char	Character	No	Character.valueOf('c')

distinciones entre primitivas y tipos referenciados:

- los primitivos siempre se nombran en minúsculas mientras que los tipos referenciados son en mayúsculas (esto es una buena practica para definir clases)
- los tipos referenciados permiten utilizar métodos que tienen incorporados mientras que los primitivos nunca tendrán ningún método incluido.
- por ultimo mencionar que a los tipo por referencia se le puede asignar ***NULL*** (lo que indica que no hay ninguna referencia a algún objeto).
- Las clases primitivas tiene sus clases wrapper (envoltura)

existen dos funciones que permite convertir cadenas a otros tipos de datos

int primitive = Integer.parseInt("123"); // convierte string en primitivos
Integer wrapper = Integer.valueOf("123"); // convierte string en objetos de las clases envolturas

todas las clases envoltorio heredan de la clase ***Number*** por lo cual vienen con métodos que permiten retornar el primitivo del tipo de dato asociado a la clase referida.

```java
Double apple = Double.valueOf("200.99");
System.out.println(apple.byteValue()); // -56 porque hay un desbordamiento del byte y 200 - 256 = -56
System.out.println(apple.intValue()); // 200
System.out.println(apple.doubleValue()); // 200.99
```

En los bloques de código existen los escapes de caracteres como \" y ***n*** el primero escapa el caracterer " para que no se considere una parte del la cadena y la segunda me permite crear un salto de linea.

Java también soporta múltiple String a través de """

```java
String mibloque = """
  "Java Study Guide"
    by scoot & Jeanne""";
```

En los bloques de texto existe el "espacio incidental" y el "espacio esencial".
El espacio esencial es parte de tu String y es importante para ti. El espacio incidental simplemente está allí para facilitar la lectura del código.

Formatting	Meaning in regular String	Meaning in text block
\"	"	"
\"""	n/a – Invalid	"""
\"\"\"	"""	"""
Space (at end of line)	Space	Ignored
\s	Two spaces (\s is a space and preserves leading space on the line)	Two spaces
\ (at end of line)	n/a – Invalid	Omits new line on that line

```
String block = """doe""";  // no compilara ya que las primeras tres comillas siempre necesitan un salto de linea.

String block = """
doe\
deer""";  // pone todo el bloque en una sola linea (respeta si hay espacios antes de deer) ya que \ evita el salto de caracteres.

String block = """
doe\n
deer
 """;  // aquí hay 4 lineas, \n crea una linea vacía ya que es un salto de linea y los """ finales son una linea final en blanco.
```

Declarando Variables

Una variable es un nombre para un espacio de memoria que almacena información.
cuando declaras una variable, necesitas definir el tipo de variable. el darle un valor a la variable se denomina "inicialización".
el identificador es el nombre a variable, métodos, clases, interfaz o paquete. Las reglas para que sean correctos son:

- identificadores deben empezar con una letra, símbolo de moneda (dólar, yuan o euro) y el símbolo guion bajo.
- identificadores pueden incluir números pero no iniciar con ellos
- un solo guion abajo no es considerado un identificador
- no se puede utilizar palabras reservadas de java

recordar que Java es **Case Sensitive** por lo que escribir una palabra con mayúsculas y minúsculas son diferentes.

TABLE 1.9 Reserved words

abstract	assert	boolean	break	byte
case	catch	char	class	const *
continue	default	do	double	else
enum	extends	final	finally	float
for	goto *	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

la palabra ***goto*** no se utiliza en Java pero igual esta reservada para algún futuro, existen otras palabras como ***null, true, false*** que tampoco pueden ser utilizadas.

aquí algunos ejemplos permitidos:

```
long okidentifier;
float $OK2Identifier;
boolean _alsoOK1d3ntifi3r;
char __SStillOkbutKnotsonice$;
```

```
int 3DPointClass; // no puede empezar con un nombre
byte hollywood@; // @ no es una palabra reservada
String *$coffee; // * no es una palabra reservada
double public; // public es una palabra reservada
short _; // un guion bajo no puede ser utilizado
```

camelCase -> se usa en métodos, variables
CamelCase -> se utiliza para clases e interfaces
snake_case -> se utiliza en constantes y enums

Declaración de múltiples variables: uno puede declarar e inicializar todos las variables que desee en una misma linea siempre y cuando sean del mismo tipo:

```
void sandFence(){
 String s1, s2;
 String s3 ="yes", s4 = "no";
} // se declararon 4 variables e inicializaron 2(s3 y s4)

void paintFence(){
 int i1, i2, i3 = 0;
} // aquí se declararon 3 pero solo se inicializo la última
```

- la inicialización de una variable (asignarle un valor) se debe dar antes de utilizarla.
- una variable local es una variable definida dentro de un constructor, método o bloque.
- las variables locales no tienen un valor por defecto por lo cual si o si deben ser inicializadas antes de ser utilizadas.

```
public void findAnswer(boolean check){
 int answer;
 int otherAnswer;
 int onlyOneBranch;
 if(check){
  onlyOneBranch = 1;
  answer = 1;
 } else {
  answer = 2;
 }
 System.out.println(answer);
 System.out.println(onlyOneBranch); // no compila por faltar un valor para onlyOneBranch en el “else”
}
```

Las variables pasadas a un constructor o método son llamados "parámetros del constructor" o "parámetros del método". Esos parámetros son como variables locales y deben ser inicializadas antes de ser utilizadas.

```
public void checkAnswer(){
 boolean value;
 findAnswer(value); // esto no compila porque "value" no se ha inicializado.
}
```

Las variables que no son variables locales se definen como variables de instancia o variables de clase.

Una **variable de instancia** a menudo llamado campo, es un valor definido dentro de una instancia específica de un objeto. asumamos una clase persona con una variable de instancia denominada "nombre" la cual es de tipo cadena. cada instancia de esta clase tendrá su propio campo "nombre" con un valor como "Pedro" o "Luis". dos instancias pueden variar este valor pero no se afectaran entre ellas.

Una v**ariable de clase** es definida a nivel de la clase por lo cual puede ser utilizada al mismo tiempo por todas las instancias de la clase ademas de no requerir una instancia de la clase para ser utilizada. para definir una variable de clase debemos de anteponer ***static*** antes del nombre de la variable. Las variables de clase no requieren ser inicializadas, estas pueden utilizar un valor por defecto entregado por Java, este puede ser ***null*** para objetos, **cero** para tipos numericos y **falso** para booleanos.

Uno puede utilizar la palabra ***var*** en vez de especificar el tipo de dato, solo para declarar variables locales en ciertas condiciones (simplemente escriba **var** en lugar del tipo primitivo o de referencia)

```
public class Zoo{
 public void whatTypeAm(){
  var name = "Hello";
  var size = 7;
 }
}
```

esto no aplica para variables de instancia por ejemplo lo siguiente arroja error:

```
public class VarKeyword{
 var tricky = "hello"; // no compila
}
```

La palabra ***var*** hace que el compilador determine el tipo de dato por uno, pero si después intentas asignar a esa variable un valor de tipo diferente se generara un error de compilación.

```
public void reassignment(){
 var number = 7;
 number = 4;
 numero = "five"; // se generar un error de compilación
}
```

al utilizar ***var*** debemos de definir e inicializar en una linea la variable, podemos hacer un salto de linea como el siguiente:

```
1 public void breaking(){
2  var silly
3   = 1;
4 }
```

el siguiente ejemplo arroja error por "question" y "answer"

```
1  public void desCompile(boolean check) {
2   var question;
3   question = 1;
4   var answer;
5   if(check){
6    answer = 2;
7   } else {
8    answer = 3;
9   }
10   System.out.println(answer);
11  }
```

```
1  public void twoTipes(){
2   int a, var b = 3; // no compila porque var detectara el numero como un real y no puede haber valores de diferente tipo en la misma linea.
3   var n = null; // esto no compila porque si bien puede referenciarse al tipo Objet, java prefiere este tipo de asignaciones.
4  }
5
6  public int addition(var a , var b){
7   return a + b; // esto no compila porque a y b son  parámetros y no variables locales.
8  }
```

```
1  package var;
2
3  public class Var{
4   public void var(){
5    var var = "var";
6   }
7   public void Var(){
8    Var var = new Var();
9   }
10  }
```

Hay una última regla de la que debes estar al tanto: "var" no es una palabra reservada y está permitido usarla como un identificador. Se considera un nombre de tipo reservado, lo cual significa que no se puede usar para definir un tipo, como una clase, interfaz o enumeración.

Por ejemplo el siguiente código compila:

URL de referencia para entender var

Las variables locales tienen un alcance que se limita como máximo al método definido. Cada bloque de código separado por {} va tener su propio scope como por ejemplo:

```
1  public void eatHi(boolean hungry){
2   if(hungry){
3    int bitesA = 1;
4   }
5   System.out.println(bitesA); // no compila porque bitesA solo vive dentro del IF
6  }
```

Revisando los alcances:

- **variables locales**: el alcance es hasta el final del bloque.
- **parámetros de métodos**: el alcance es durante la duración del método.
- **variables de instancia**: el alcances es desde la declaración hasta que el objeto es escogido por el recolector de basura.
- **variables de clase**: el alcance es desde la declaración hasta la finalización del programa.

Todos los objetos de tu programa java son almacenados en la **memory heap** que es un elemento de la JVM.

El **garbage collector** es un proceso que automáticamente depura los elementos que no se necesitan del **memory heap**. En Java un elemento elegible para el **garbage collector** son objetos que no están siendo accedido desde ningún lugar del programa.

Java tiene un método interno ***System.gc()*** el cual puede servir a para que uno solicite a Java realizar la ejecución del **garbage collector** pero este se reserva el derecho a hacerlo.

El **garbage collector** censa los objetos hasta detectar que ya no son necesarios para depurarlos bajo 2 condiciones:

- El objeto ya no tiene ninguna referencia que apunte a él.
- Todas las referencias al objeto han quedado fuera de alcance.

La referencia es una variable que tiene un nombre y puede ser utilizada. una referencia puede ser asignada a otra, pasada a un método, ser retornada por un método. todas las referencias tienen el mismo tamaño sin importar el tipo.

Un objeto reside en el heap y no tiene un nombre. Por lo tanto, no hay forma de acceder a un objeto excepto a través de una referencia. Los objetos vienen en todas las formas y tamaños diferentes y consumen cantidades variables de memoria. Un objeto 1 no puede ser asignado a otro objeto, y un objeto no puede ser pasado a un método o devuelto desde un método. Es el objeto el que se recolecta como basura, no su referencia.

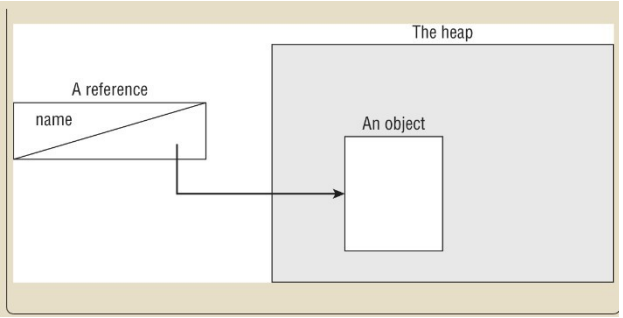


Figura 3: referencia entre la variable y el objeto

analizando el siguiente código para identificar las referencias y objetos:
vamos a concluir que el objeto "a" sera depurado por el **garbage collector** y el objeto "b" sera elegible cuando el método concluya.

```
1  public class Scope{
2   public static void main(String[] args){
3    String one, two; // solo definimos 2 nombres
4    one = new String("a"); // se crea el primer objeto en el heap y se enlaza a "one"
5    two = new String("b"); // se crea el segundo objeto en el heap y se enlaza a "two"
6    one = two; // ahora "one" apunta al objeto "b"
7    String three = one; // se crea un nuevo nombre y apunta al objeto "b"
8    one = null; // por ultimo hacemos que el nombre "one" no apunte a nada
9   }
10 }
```

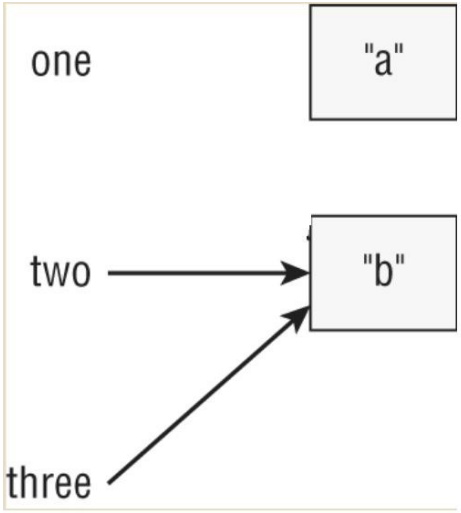


Figura 4: referencia del ejemplo