INTRODUCCIÓN

Encapsulación.

La encapsulación es un mecanismo de programación que combina el código con los datos que manipula, al tiempo que los proteger de interferencias externas. El código y los datos se pueden vincular de manera que se cree una caja negra independiente. Al vincularlos de esta forma se crea un objeto. Es decir, un objeto es el dispositivo que permite la encapsulación.

En un objeto, el código y/o los datos pueden ser privados o públicos. El código o los datos privados son conocidos y solo se puede acceder a los mismos desde otra parte del objeto. Es decir, no se puede acceder desde un programa externo al objeto. Cuando el código o los datos son públicos, otras partes del programa pueden acceder, aunque se definan en un objeto. Por lo general, las partes públicas de un objeto se usan para proporcionar una interfaz controlada a los elementos privados del objeto.

La unidad básica de encapsulación en java es la clase.

Polimorfismo.

Imagine una pila (un lista de tipo primero en entrar, último en salir). Puede tener un programa que requiera tres tipos de pilas distintas. Usa se usa para valores enteros, otra para valores de coma flotante y otras para caracteres. El algoritmo que implementa cada pila es el mismo, aunque los datos almacenados difieran. En lenguajes no orientas a objetos tendría que crear tres conjuntos diferentes de rutinas de pila, cada una con un nombre. Sin embargo, gracias al polimorfismo, en Java puede crear un conjunto general de rutinas de pila que funcione en los tres casos. De este modo, si sabe usar una pila, puede usarlas todas.

Herencia.

La herencia es el proceso mediante el que un objeto puede adquirir las propiedades de otro. Una clase hereda todas las cualidades de las clases anteriores y define solamente las que la hacen única. Sin el uso de jerarquías, cada objeto tendría que definir explícitamente todas sus características. Gracias a la herencia, el objeto solo tiene que definir las cualidades que lo hacen único dentro de la clase. Puede heredar sus atributos generales de su principal. Por tanto, el mecanismo de herencia permite que un objeto sea una instancia específica de un caso más general.

[HelloWorld.java]

```java
class HelloWorld {

    // Un programa de java comienza con la invocación de main().
    public static void main(String args[]){
        System.out.println("Hello, world.");
    }
}
```

En Java, un archivo se denomina oficialmente unidad de compilación. Un archivo contiene, entre otros elementos, una o varias clases. Por convención, el nombre de un archivo y el de su clase principal deben coincidir; en Java todo el código se guarda en una clase.

El programa comienza con:

```
class hello_world {
```

Usa la palabra clave 'class' para declarar que se define una nueva clase. La clase es la unidad de encapsulación básica de java. 'hello_world' es el nombre de la clase (y también del archivo al que pertenece). Los elementos entre las llaves son los miembros de la clase. Lo que es necesario recordar es que en Java, la actividad del programa se produce en una clase, motivo por el que todos los programas son orientados a objetos.

La siguiente línea de código:

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

inicia el método main(). En Java las subrutinas se llaman métodos. En esta línea comienza la ejecución del programa. Todas las aplicaciones de Java se inician mediante la invocación de main().

'public' es una modificador de acceso, que determina cómo otras partes del programa pueden acceder a los miembros de la clase. Cuando un miembro de clase se precede de public, se puede acceder a dicho miembro desde código externo a la clase en la que se declara. Lo contrario es private, que evita que el miembro se use con código definido fuera de su clase. En este caso debe declararse main() como public, ya que debe invocarse por código externo a la clase al iniciar el programa.

'static', esta palabra clave permite invocar main() antes de crear un objeto de la clase. Es necesario, ya que Java invoca a main() antes de crear objetos. 'void' indica que main() no devuelve valores. 'main()' es el método invocado al iniciar una aplicación Java. En main() solo hay un parámetro, String args[], que declara el parámetro args[], una matriz de objetos de objetos tipo String.

En la siguiente línea:

```
System.out.println("Hello, world.");
```

'System' es una clase predefinida que ofrece acceso al sistema y out es el flujo de salida conectado a la consola. 'System.out' es, por tanto, un objeto que encapsula el resultado de la consola.

'println' es un método que se pasa al objeto anterior con el argumento "Hello, world", Muestra esta cadena en pantalla.

[ExampleVars.java]

```
class ExampleVars {
   public static void main(String args[]){
       int var1;
       int var2;
       var1 = 1024;

       System.out.println("var1 contiene " + var1);     /* Imprime: var1
                                                           contiene 1024 */
       var2 = var1/2;

       System.out.print("var2 contiene var1/2: ");
       System.out.println(var2);    // Imprime: var2 contiene var1/2: 512
   }
}
```

El signo '+' hace que el valor de var1 se muestre después de la cadena que lo precede, con este operador '+' puede combinar todos los elementos que desee en la misma instrucción println().

El método print es similar a println, pero no muestra una nueva línea tras la invocación.

[IfDemo.java]

```
class IfDemo {
    public static void main(String args[]){
        int a, b, c;

        a = 2;
        b = 3;

        if(a < b) System.out.println("a es menor que b");
        if (a == b) System.out.println("Esto no se imprime");

        System.out.println();

        c = a - b;

        System.out.println("c contiene el valor -1");

        if(c >= 0) System.out.println("c no es negativo");

        if(c < 0) System.out.println("c es negatigo");

        System.out.println();

        c = b - a;

        System.out.println("c contiene el valor 1");

        if(c >= 0) System.out.println("c no es negativo");

        if(c < 0) System.out.println("c es negatigo");
    }
}
```

[ForDemo.java]

```
class ForDemo {
    public static void main(String args[]){
        double gallons, liters;
        int counter = 0;

        for(gallons = 1; gallons <= 100; gallons++){
            liters = gallons * 3.7854;
            System.out.println(gallons + " galones son " + liters + " litros.");
            counter++;

            if(counter == 10){  // Cada 10 líneas imprimir una en blanco.
                System.out.println();
                counter = 0;
            }
        }
    }
}
```

Los programas mostrados en esta introducción, usan dos métodos de Java, print() y println(). Estos métodos son miembros de la clase System, una clase predefinida que Java incluye de forma automática en sus programas. Java es una combinación del propio lenguaje más sus clases estándar. Las bibliotecas de clases proporcionan la funcionalidad de Java.

TIPOS DE DATOS Y OPERADORES

Números.

Los tipos de datos en Java son tipos fuertes, es decir el compilador comprueba la compatibilidad de los tipos en todas las operaciones. Las operaciones incompatibles no se compilan. Para realizar la comprobación de tipos, todas las variables, expresiones y valores tienen un tipo. No existe el concepto de variable sin tipo, por ejemplo. Una operación permitida en un tipo puede no estarlo en otro.

Java tiene dos categorías de tipos integrados: orientados a objetos y no orientas a objetos. Los tipos orientados a objetos se definen en clases. Encontramos ocho tipos de datos primitivos elementales. El término primitivo se usa para indicar que estos tipos no son objetos en un sentido orienta a objetos, sino valores binarios estándar. Los demás tipos en Java se crean a partir de los primitivos. Los tipos primitivos son:

boolean	true/false	
byte	Entero de 8 bits	-128 a 127
char	Carácter	
double	Coma flotante de precisión doble (64 bits)	
float	Coma flotante de precisión simple (32 bits)	
int	Entero (32 bits)	...
long	Entero largo (64 bits)	...
short	Entero corto (16 bits)	-32.768 a 32.767

Java no admite enteros sin signo (solo positivos), al diseñar el lenguaje no se contemplo esta posibilidad, se estimó no oportuna.

El tipo de entero más usado en int. Se usa para controlar bucles, indexar matrices y realizar operaciones matemáticas generales.

La variables tipo byte son especialmente útiles para trabajar con datos binarios sin procesar que pueden no ser compatibles directamente con los tipos de Java.

Las variables tipo short sirven cuando no es necesario el intervalo que ofrece int.

De los tipos en coma flotante double es el más usado.

[Hypot.java]

```
class Hypot {
    public static void main(String args[]){
    double x, y, z;

    x = 3;
    y = 4;
    z = Math.sqrt(x*x + y*y);

    System.out.println("La hipotenusa es: " + z);
    }
}
```

En el ejemplo anterior, sqrt() es miembro de la clase estándar Math. En su invocación se precede del nombre Math. Aunque no todos los métodos estándar se invocan especificando primero el nombre de su clase, algunos si lo hacen.

Caracteres.

En Java los caracteres no son cantidades de 8 bits. Java usa Unicode (conjunto de caracteres que puede representar todos los signos de todos los idiomas). char es un tipo sin signo de 16 bits, con un intervalo de 0 a 65.536. ASCII es un subconjunto de Unicode. Ejemplo (dar un valor a un char):

```
char  ch;
ch = 'x';
```

Las variables de caracteres se pueden procesar como enteros. Como char es un tipo sin signo de 16 bits, se puede usar para realizar operaciones aritméticas. Por ejemplo:

[CharArithDemo.java]

```
class CharArithDemo {
    public static void main(String args[]){
        char ch;
        ch = 'x';

        System.out.println("ch tiene el valor " + ch);

        ch++;    // Suma 1 a ch.
        System.out.println("ch tiene ahora el valor de " + ch);

        ch = 122;  // Asigna el valor 122 (z) a ch.
        System.out.println("ch tiene ahora el valor de " + ch);
    }
}
```

Tipo bolean.

[BoolDemo.java]

```
class BoolDemo {
    public static void main(String args[]){
        boolean b;

        b = false;
        System.out.println(" b es " + b);
        b = true;
        System.out.println(" b es " + b);

        // Un valor boolean puede controlar la instrucción if.
        if(b) System.out.println("Esto se ejecuta");

        // El resultado de un operador relacional es un valor boolean.
        System.out.println("10 > 9 es " + (10 > 9));
    }
}
```

Observar que al representar el valor boolean con println() se muestra 'true' o 'false'. Además el valor de una variable boolean es suficiente para controlar la instrucción if, es decir no es necesario escribir:

```
if(b == true);
```

Por último observar que el resultado de un operador relacional, como '<' es un boolean; la expresión (10 > 9) muestra 'true'.

Literales.

En Java, un literal es un valor fijo representado en formato legible para los humanos, por ejemplo '100' es un literal, también se denominan constantes. De forma predeterminada los literales enteros son de tipo int, si desea especificar un literal long, debe adjunto l o L. Por ejemplo, 12 es int, pero 12L es long. Del mismo modo, de forma predeterminada, los literales de coma flotante son de tipo double. Para especificar un literal float, debe añadir F o f a la constante, como 10.19F.

Aunque los literales enteros crean un valor int de forma predeterminada, pueden asignarse a variables de tipo char, byte o short, siempre que el valor asignado se pueda representar por uno de estos tipos de destino. Se pueden añadir guiones bajos a un literal entero o flotante, de este modo se facilita la lectura de los valores formados por varios dígitos. Al compilar el literal, se descartan los guiones bajos, por ejemplo:

```
123_45_1234
```

representaría al número 123.45.1234. Esto resulta especialmente válido con por ejemplo ID de clientes o códigos de estado que suelen estar formados por subgrupos de dígitos.

Literales hexadecimales, octales y binarios.

```
hex = 0xFF;        // 255 en decimal.
Oct = 011;         // 9 en decimal.
bin = 0b100;       // 12 en decimal.
```

Literales de cadena.

Una cadena es un conjunto de caracteres entre comillas dobles.

IMPORTANTE: No debe confundirse cadenas con caracteres. No es lo mismo "k" que 'k'. Un literal de carácter representa una sola letra de tipo char. Una cadena con una sola letra sigue siendo una cadena. Aunque las cadenas están formadas por caracteres no son del mismo tipo.

Variables.

El tipo de una variable no puede cambiar mientras exista. Es decir, por ejemplo, una variable int no se puede convertir en char.

Al declarar dos o más variables del mismo tipo con un alista separada por comas, puede asignar un valor inicial a una o varias de ellas:

```
int a, b = 8, c = 19, d; // b y c tienen inicializaciones.
```

Inicialización dinámica: estamos ante una inicialización dinámica cuando, por ejemplo, una variable depende de otra, como aquí:

```
double radius = 4, height = 5;
double volume = 3,1416 * radius * radius * height;
```

Aquí se declaran tres variables y una de ellas se inicializa dinámicamente.

Una expresión de inicialización puede contener cualquier elemento válido en el momento de la inicialización, incluyendo invocaciones de métodos, otras variables o literales.

Ámbito y duración de variables.

Java permite declarar variables en cualquier bloque (por ejemplo antes de main()). Un bloque define un ámbito, cada vez que se crea un nuevo bloque {}, se crea con el un ámbito nuevo. El ámbito determina que objetos son visibles para otras partes del programa y también la duración de dichos objetos. Otros lenguajes definen los ámbitos local y global. Java los admite, pero no son la mejor opción para categorizar los ámbitos de Java. Los más importantes son los definidos por clases y métodos.

El ámbito definido por un método comienza con su llave de apertura. No obstante si ese método tiene parámetros también se incluyen en su ámbito.

[ScopeDemo.java]

```
class ScopeDemo {
    public static void main(String args[]){
        int x;  // Conocido para todo el ámbito de main

        x = 10;
        if(x == 10){     // Iniciar nuevo ámbito
            int y = 20; // Solo es conocido para este bloque

            // x e y se conocen aquí

            System.out.println("x e y: " + x + " " + y);
            x = y * 2;
        }

        //y = 100;  // Error. y es desconocido aquí.

        //x sigue siendo conocido
        System.out.println("x es " + x);     // Imprime: x es 40

    }
}
```

Como regla general , las variables declaradas en un ámbito no son accesibles para el código definido fuera de ese ámbito. Por ello, al declarar una variable en un ámbito, se localiza y se protege de acceso sin autorización y modificaciones. En realidad las reglas de ámbito constituyen la base de la encapsulación.

Los ámbitos se pueden anidar. El ámbito externo contiene al interno y un objeto declarado en el ámbito externo será visible para el código del ámbito interno, pero no a la inversa: los objetos declarados en el ámbito interno no son visibles en el exterior del mismo.

Si se define (declara) una variable al final de un bloque no servirá de nada, ya que ningún código podrá acceder a la misma. Las variables de los métodos se definen al principio del código del método, así están disponible para todo el método.

Una variable no mantiene su valor tras salir de su ámbito. Por tanto las variables definidas en un método no conservan sus valores entre invocaciones del mismo. Además una variable definida en un bloque, pierde su valor al salir del bloque. Es decir, IMPORTANTE: la duración de una variable depende de su ámbito. Si la declaración de una variable incluye un inicializador, dicha variable se vuelve a inicializar cada vez que se acceda al bloque en el que se declara.

Las reglas del ámbito de Java tiene una peculiaridad. Aunque los bloques se pueden anidar, las variables declaradas en un ámbito interior no pueden tener le mismo nombre que las declaradas por el ámbito contenedor. Por ejemplo, el siguiente código no sería correcto:

```
int count;
for(count = 0; count < 10; count++){
    ...
    int count;           // Incorrecto. Ya se ha declarado count antes.
    for(count = 0; ...){
}
```

Asignaciones abreviadas.

En las siguientes expresiones ambas asignan a x el valor de 'x + 10' o el valor de 'x - 10'.

```
x = x + 10      -->     x += 10
x = x - 10      -->     x -= 10
```

La abreviatura funciona en todos los operadores binarios en Java, es decir, aquellos que requieren dos operandos, el formato general de la expresión es el siguiente:

```
var op= expresión;
```

Como estos operadores combinan una operación con una asignación, suelen denominarse operadores de asignación compuestos.

Conversiones de tipos.

Una conversión explícita tiene el siguiente formato:

```
(tipo-destino) expresión;
```

Por ejemplo:

```
double x, y;
...
(int) (x/y); // se convierte el resultado de la expresión x/y a int.
```

[ConversionDemo.java]

```
class ConversionDemo {
    public static void main(String args[]){
        double x, y;
        byte b;
        int i;
        char ch;

        x = 10.0;
        y = 3.0;
```

```
        i = (int) (x / y); // Convertir double a int. Se produce recorte.
        System.out.println("La conversión a entero de x/y es: " + i);

        i = 100;
        b = (byte) i; // No se pierde información. byte almacena el valor 100.
        System.out.println("El valor de b es: " + b);

        i = 257;
        b = (byte) i; // Ss pierde información. byte no almacena el valor 257.
        System.out.println("El valor de b es: " + b);
        b = 88;     // Código ASCII para 'x'.
        ch = (char) b; // Conversión entre tipos.
        System.out.println("ch: " + ch);
    }
}
```

INSTRUCCIONES DE CONTROL

Introducir caracteres desde le teclado.

Para leer un carácter desde le teclado usamos el método read() de system, 'System.in.read()'. System.in es el objeto de entrada conectado al teclado. El método read() espera a que el usuario pulse una tecla y después devuelve el resultado. El carácter se devuelve como entero, por lo que hay que convertirlo a char antes de asignarlo a una variable tipo char.

De forma predeterminada, las entradas de consola se almacenan en un búfer antes de que el programa los lea. En este caso el búfer contiene una línea completa de texto y hay que pulsar 'enter' antes de enviar los caracteres que se escriban al programa.

El siguiente programa lee un carácter desde el teclado:

[KbIn.java]

```
class KbIn {
    public static void main(String args[])
        throws java.io.IOException {

            char ch;
            System.out.print("Pulse una tecla seguida de 'intro': ");

            ch = (char) System.in.read();  // Leer un carácter desde el teclado.

            System.out.println("La tecla pulsada es: " + ch);
    }
}
```

El programa comienza con la expresión 'throws java.io.IOException'. Se debe especificar esta clausula para procesar errores de entrada; forma parte del mecanismo de control de excepciones de Java.

En ocasiones, el hecho de que System.in se almacene en el búfer puede dar problemas. Al pulsar 'intro', se añade al flujo de entrada una secuencia de retorno de carro y nueva línea. Además estos caracteres se mantienen en el búfer de entrada hasta que los lea. Por ello, en lagunas aplicaciones tendrá que eliminarlos (leerlos) antes de la siguiente operación de entrada. Esto lo veremos más adelante.

Ejemplos de expresiones de control: if-else-if y switch.

El siguiente programa ilustra el uso de if-else-if.

[Ladder.java]

```
class Ladder {
    public static void main(String args[]){
        for(int x=0; x<6; x++){
            if(x==1)
                System.out.println("x es uno");
            else if(x==2)
                System.out.println("x es dos");
            else if(x==3)
```

```
                System.out.println("x es tres");
            else if(x==4)
                System.out.println("x es cuatro");
            // Instrucción predeterminada.
            else System.out.println("x no está entre uno y cuatro");
        }
    }
}
```

El siguiente programa ilustra el uso de switch.

[SwitchDemo.java]

```
class SwitchDemo {
    public static void main (String args[]){
        for(int i=0; i<10; i++)
            switch(i){
                case 0:
                    System.out.println(" i es cero");
                    break;
                case 1:
                    System.out.println("i es uno");
                    break;
                case 2:
                    System.out.println("i es dos");
                    break;
                case 3:
                    System.out.println("i es tres");
                    break;
                case 4:
                    System.out.println("i es cuatro");
                    break;
                default:
                    System.out.println("i es cinco o mayor");
            }
    }
}
```

En el ejemplo anterior, si i es cinco o superior, no coinciden instrucciones case, por lo que se ejecuta la instrucción default.

La instrucción break es opcional. Si una instrucción break no finaliza la secuencia de instrucciones asociadas a case, se ejecutan todas las instrucciones tras case hasta detectar break o el final de switch. Por ejemplo, en el siguiente programa la salida es:

```
i es menor que uno
i es menor que dos
i es menor que tres
i es menor que cuatro
i es menor que cinco

i es menor que dos
i es menor que tres
i es menor que cuatro
i es menor que cinco

i es menor que tres
```

```
      i es menor que cuatro
      i es menor que cinco

      i es menor que cuatro
      i es menor que cinco

      i es menor que cinco
```

[NoBreak.java]

```
class NoBreak {
    public static void main (String args[]){
        for(int i=0; i<=5; i++)
            switch(i){
                case 0:
                    System.out.println(" i es menor que uno");
                case 1:
                    System.out.println("i es menor que dos");
                case 2:
                    System.out.println("i es menor que tres");
                case 3:
                    System.out.println("i es menor que cuatro");
                case 4:
                    System.out.println("i es menor que cinco");
            }

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

Como puede verse, la ejecución prosigue hasta la siguiente constante case si no hay instrucción break.

También es posible usar instrucciones case vacías, como aquí:

```
      switch(i){
          case 1:
          case 2:
          case 3: System.out.println("i es 1, dos o tres");
              break;
          case 4: System.out.println("i es cuatro");
              break;
      }
```

En este ejemplo, si i tiene el valor 1, 2 o 3, se ejecuta la primera instrucción println(). Si es 4, se ejecuta la segunda instrucción println().

El bucle for.

for es una de las instrucciones más versátiles de Java ya que permite distintas variantes. Por ejemplo se pueden usar múltiples variables de control, como en el siguiente programa:

[VarsControlFor.java]

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

```
        int i, j;
        for (i=0, j=10; i < j; i++, j--)
            System.out.println("i y j: " + i + " " + j);
    }

}
```

Puede usar todas las instrucciones de inicialización e iteración que desee, pero en la práctica más de dos o tres dificultan el bucle for.

La expresión que controla el bucle puede ser cualquier expresión booleana. No es necesario usar la variable de control del bucle. Por ejemplo:

[ForTest.java]

```
class ForTest {
    public static void main(String args[])
        throws java.io.IOException{

        System.out.println("Pulse 'intro' para comenzar. " +
                           "Pulse 's' para detener.");

        for(int i = 0; (char) System.in.read() != 's'; i++)
            System.out.println("Pase #" + i);
    }
}
```

Las partes de inicialización, condición o control del bucle for se pueden dejar en blanco, creando así variantes interesantes del bucle. Por ejemplo:

[EmptyFor.java]

```
class EmptyFor {
    public static void main(String args[]){
        int i = 0;  // Se extrae la inicialización del bucle.
        for( ; i < 10; ){
            System.out.println("Pase #" + i);
            i++;    // Se incrementa la variable de control del bucle.
        }
    }
}
```

La ubicación de la variable de inicialización fuera del bucle, como en este caso, normalmente, suele hacerse cuando el valor inicial se deriva de un proceso complejo que no puede incluirse dentro de la instrucción for. Veamos ahora un ejemplo de un bucle for sin cuerpo:

[EmptyBodyFor.java]

```
class EmptyBodyFor {
    public static void main(String args[]){
        int i;
        int sum = 0;

        // sumar los números hasta 5
        for(i=1; i <= 5; sum += i++) // sum += i++;  <-- sum = sum + i; i++;
```

```
            ;   // Este bucle no tiene cuerpo
        System.out.println("La suma es: " + sum);    // Imprime: La suma es 15
    }
}
```

La expresión 'sum += i++' es una expresión habitual en programas profesionales y resulta fácil de entender si se divide en sus componentes como muestra el comentario del ejemplo.

Ejemplo de uso de bucle while/do-while.

[DoWhile.java]

```
class DoWhile {
    public static void main(String args[])
        throws java.io.IOException{

        char ch, ignore, answer = 'K';

        do{
            System.out.println("He pensado una letra de la A a la Z.");
            System.out.println("¿Puedes adivinarla?: ");

            // leer un carácter
            ch = (char) System.in.read();

            // descartar otros caracteres del búfer de entrada.
            do{
                ignore = (char) System.in.read();
            } while(ignore != '\n');

            if(ch == answer) System.out.println("¡La has adivinado!");
            else{
                System.out.print("Lo siento, ");
                if(ch < answer) System.out.println("letra demasiado baja.");
                else System.out.println("letra demasiado alta.");
                System.out.println("Prueba otra vez");
            }

        } while(answer != ch);
    }
}
```

Usar break como goto.

Además de sus usos con switch y bucles, break se puede usar como forma civilizada de la instrucción goto, ausente en java por ser una forma desestructurada de alterar el flujo de ejecución de un programa. Para salir de un conjunto anidado de bucles, Java define define una forma ampliada de break, se aplica a cualquier bloque. Y es más puede especificar dónde se reanudará la ejecución, ya que esta forma de break funciona con una etiqueta. Es decir, break ofrece las ventajas de goto, pero sin sus problemas.

El formato general es el siguiente:

```
    break etiqueta;
```

Para asignar un nombre a un bloque, incluya la etiqueta al inicio del mismo. El bloque etiquetado puede ser un bloque independiente o una instrucción con un bloque como destino. Una etiqueta es cualquier identificador válido de Java seguido de dos puntos. Tras etiquetar un bloque, puede usar esta etiqueta como destino de una instrucción break. Por ejemplo:

[BreakToLabel.java]

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

        for (int i = 1; i < 4; i++){
            one:{
                two:{
                    three:{
                            System.out.println("i es " +i);
                            if(i == 1) break one; // salir a una etiqueta.
                            if(i == 2) break two;
                            if(i == 3) break three;
                            // nunca se alcanza
                            System.out.println("NO se imprimirá");
                    }
                    System.out.println("Después del bloque three");
                }
                System.out.println("Después del bloque two");
            }
            System.out.println("Después del bloque one");
        }
        System.out.println("Después del bloque for");
    }
}
```

La salida por consola del anterior ejemplo será la siguiente:

```
i es 1
Después del bloque one

i es 2
Después del bloque two
Después del bloque one

i es 3
Después del bloque three
Después del bloque two
Después del bloque one
Después del bloque for
```

Cuando i es 1, la primera instrucción if es satisfactoria y break finaliza el bloque de código definido por la etiqueta one, se imprime "Después del bloque uno". El código funciona sucesivamente así.

Recuerde que no puede salir de etiquetas no definidas para un bloque contenedor. El siguiente programa no es válido y no se compilaría:

[BreakErr.java]

```
// Este programa contiene un error
class BreakErr {
    public static void main(String args[]){
        one: for(int i=0; i<3; i++){
            System.out.print("Pase " + i + ": ");
        }

        for(int j=0; j<100; j++){
            if(j == 10) break one; // Error: The label one is missing.
            System.out.print(j + " ");
        }
    }
}
```

Utilizar continue.

Se puede forzar una iteración de un bucle e ignorar su estructura de control normal por medio de continue. Esta instrucción fuerza la siguiente iteración del bucle e ignora el código comprendido entra la instrucción y la expresión condicional que controla el bucle.

Como sucede con break, continue puede especificar una etiqueta para describir el bucle contenedor en el que continuar.

En bucles while y do-while, la instrucción continue hace que el control pase directamente a la expresión condicional y después prosigue el proceso el bucle. En el caso del bucle for, se evalúa la expresión de iteración del bucle, se ejecuta la expresión condicional y después el bucle continúa.

[Continue.java]

```
class Continue {
    public static void main(String args[]){
        // imprimir números pares entre 0 y 100
        for(int i=0; i<=100; i++){
            if((i%2) != 0) continue; //iterar
            System.out.println(i);
        }
    }
}
```

[ContinueToLabel.java]

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

        outerloop:
        for (int i=1; i<10; i++){
            System.out.print("\nBucle externo pase " + i + ", Bucle interno: ");
            for(int j=1; j<10; j++){
                if(j == 5) continue outerloop; // continuar bucle externo
                System.out.print(j);
            }
        }
    }
}
```

En el ejemplo anterior, al ejecutar continue, el control pasa al bucle externo y se ignorar el resto del bucle interno. Imprime lo siguiente:

```
Bucle externo pase 1, Bucle interno: 1234
Bucle externo pase 2, Bucle interno: 1234
Bucle externo pase 3, Bucle interno: 1234
Bucle externo pase 4, Bucle interno: 1234
Bucle externo pase 5, Bucle interno: 1234
Bucle externo pase 6, Bucle interno: 1234
Bucle externo pase 7, Bucle interno: 1234
Bucle externo pase 8, Bucle interno: 1234
Bucle externo pase 9, Bucle interno: 1234
```

Los usos positivos de continue son escasos, pero a veces se necesita una iteración temprana y en estos casos continue constituye una forma de conseguirla.

CLASES, OBJETOS Y MÉTODOS

Clases: aspecto esenciales.

En una clase se definen datos y código que actúa sobre dichos datos. El código se incluye en métodos.

Toda la actividad de un programa en Java se produce en una clase; hemos estado usando clases del principio de estos apuntes de Java.

Una clase es una plantilla que define la forma de un objeto. Especifica los datos y el código que actúa sobre ellos. Una clase es, en definitiva, un conjunto de planos que especifican cómo crear un objeto. Conviene recordar que una clase es una abstracción lógica, hasta que no se crea un objeto de la misma, no existe una representación física de la clase en memoria.

La creación de un objeto de la clase recibe el nombre de instanciación de la clase.

Al definir una clase, se declara su forma y naturaleza exactas. Para ello se especifican las variables de instancia que contiene y los métodos que pueden actuar sobre ellas. Aunque las clases muy simples pueden contener solo métodos o solo variables de instancia, la mayoría de las clases contiene ambos elementos.

Una clase se crea mediante la palabra 'class': *

```
class NombreDeClase {
    // declarar variables de instancia
    tipo var1;
    ...
    tipo varN;

    // declarar métodos
    tipo método1(parámetros){
        // cuerpo método
    }
    ...
    tipo métodoN(parámetros){
        // cuerpo método
    }
}
```

Una clase bien diseñada agrupa información lógica vinculada; por ejemplo una clase que almacene nombres y números de teléfono, no almacenará otros datos.

El formato general de una clase no especifica el método main(). Un método main() solo es necesario si la clase es el punto de partida de un programa. además, algunos tipos de aplicaciones de Java no requieren main().

Construyamos una clase inicial:

```
class Vehicle {
      int passengers;     // número de pasajeros
      int fuelcap;        // capacidad de combustible
      int mpg;            // consumo de combustible
}
```

Una definición de clase crea un nuevo tipo de datos. Es este caso el tipo 'Vehicle', usaremos este nombre para declarar objetos tipo vehicle. IMPORTANTE: una declaración de clase es solo un descripción de tipos, no crea un objeto. Por tanto el código anterior no crea objetos de tipo Vehicle; para crear un objeto

debemos hacer como sigue:

```
Vehicle minivan = new Vehicle();// Crea objeto Vehicle con nombre minivan.
```

Tras ejecutar esta instrucción, minivan sera una instancia de Vehicle y existirá en el programa. Todas las instancias creadas de Vehicle, como minivan, tendrán su propia copia de las variables passengers, fuelcap y mpg. Para acceder a estas variables, debe usar el operador punto '.', que vincula el nombre del objeto al nombre de un miembro. El formato general de este operador es:

```
objeto.miembro
```

Por ejemplo, para signar a la variable fuelcap de minivan el valor 16 se usa:

```
minivan.fuelcap = 16;
```

Por lo general se puede usar el operador punto '.' para acceder a variables de instancia y métodos.

[VehicleClass.java]

```
 /*Un programa que usa la clase Vehicle.
   Asignar el nombre VehicleClass.java a este archivo.*/

class Vehicle {
   int passengers;
   int fuelcap;
   int mpg;
}

// Esta clase declara un objeto tipo Vehicle.

class VehicleClass {
   public static void main(String args[]){
       Vehicle minivan = new Vehicle();
       int range;

       // asignar valores a los campos de minivan
       minivan.passengers = 7; // uso del operador . para acceder a un miembro.
       minivan.fuelcap = 16;
       minivan.mpg = 21;

       // calcular la autonomía del vehículo.
       range = minivan.fuelcap * minivan.mpg;
       System.out.println("Minivan puede transportar a " + minivan.passengers +
                          " con una capacidad de " + range);
   }
}
```

Debe asignar el nombre VehicleClass.java al archivo que contenga el programa, ya que el método main() se encuentra en la clase VehicleClass, no en la clase Vehicle. Al compilar el programa se crean dos archivos . Class, uno para Vehicle y otro para VehicleClass. El compilador de java incluye automáticamente cada clase en su propio archivo .class. No es necesario que Vehicle y VehicleClass se encuentren en el mismo archivo fuente, se pueden incluir en archivos diferentes como Vehicle.java y VehicleClass.java, respectivamente.

La línea:

```
Vehicle minivan = new Vehicle();
```

es equivalente a:

```
Vehicle minivan;           // declarar referencia al objeto (Declaración).
minivan = new Vehicle();   // asignar un objeto Vehicle (Asignación).
```

El operador new asigna dinámicamente (es decir, en tiempo de ejecución) memoria par aun objeto y devuelve una referencia al mismo. Esta referencia es mas o menos la dirección en memoria del objeto asignado por new, que después se almacena en una variable. Así pues en Java, todos los objetos de clase deben asignarse de forma dinámica.

En el ejemplo anterior, la primera línea declara minivan como referencia a un objeto de tipo Vehicle. Por tanto, minivan es una variable que puede hacer referencia a un objeto, pero no es un objeto todavía. La siguiente línea crea un nuevo objeto Vehicle y asigna a minivan una referencia al mismo. Ahora minivan está vinculada a un objeto.

Métodos.

Los métodos son subrutinas que manipulan los datos definidos por la clase y, en muchas ocasiones, permiten acceder a dichos datos. Además, otras partes de un programa interactúan con la clase a través de sus métodos. En un código bien escrito, cada método realiza una sola tarea. El formato general de un método es el siguiente:

```
tipo-devuelto nombre (lista-de-argumentos) {
      //cuerpo del método
}
```

tipo-devuelto especifica el tipo de datos que devuelve el método, que puede ser cualquier tipo válido, incluidos los tipos de clase que cree. Si un método no devuelve valores su tipo debe ser 'void'.

Por ejemplo, para añadir un método a la clase Vehicle, debemos especificarlo en la propia clase. La siguiente versión de vehicle contiene el método range() que muestra la autonomía del vehículo:

[Method.java]

```
class Vehicle {
   int passengers;
   int fuelcap;
   int mpg;

   // Mostrar la autonomía.
   // El método range() se incluye en la clase vehículo.
   // [*] fuelcap y mpg se usan directamente, sin el operador '.'.
   void range() {
       System.out.println("La autonomía es " + fuelcap * mpg);
   }
}

class Method {
   public static void main(String args[]) {
       Vehicle minivan = new Vehicle();
```

```
        Vehicle sportscar = new Vehicle();

        minivan.fuelcap = 16;
        minivan.passengers = 7;
        minivan.mpg = 21;

        sportscar.passengers = 2;
        sportscar.fuelcap = 14;
        sportscar.mpg = 12;

        System.out.print("Minivan trasnporta" + minivan.passengers + ". ");
        minivan.range();  // Mostrar autonomía de Minivan.
        System.out.print("Sportscar trasnporta" + sportscar.passengers + ". ");
        sportscar.range();
    }
}
```

La instrucción 'minivan.range();' invoca al método range() con respecto al objeto minivan. Al invocar un método, el control del programa se transfiere al método. Cuando el método concluye, el control regresa al invocador y la ejecución del programa se reanuda en la siguiente línea de código tras la invocación.

[*] Cuando un método usa una variable de instancia definida por su clase, lo hace directamente, sin hacer referencia de forma explícita a un objeto y sin usar el operador punto.

Constructores.

En el ejemplo anterior la variables de instancia de cada tipo Vehicle debían establecerse manualmente por medio de una secuencia de instrucciones, asignando un valor a cada variable.

Este enfoque nunca se usaría en un código profesional. Existe una forma más sencilla de realizar esta tarea, con un constructor.

Un constructor inicializa un objeto al crearlo, tiene el mismo nombre que su clase y es sintácticamente similar a un método. Sin embargo los constructores carecen de un tipo de devolución explícito. Por lo general los constructores se usan para asignar valores iniciales a las variables de instancia definidas por la clase.

Todas las clases tienen constructores, los defina o no, ya que Java ofrece automáticamente un constructor predeterminado que inicializa todas las variables miembro en sus valores predeterminados, que son 0, null y false para numéricos, tipos de referencia y tipos boolean, respectivamente. No obstante, tras definir su propio constructor, el predeterminado deja de usarse.

[Constructor.java]

```
class MyClass {
    int x;

    // El constructor de MyClass
    MyClass() {
        x = 10;
    }
}
```

```
class Constructor {
   public static void main(String args[]) {
       MyClass t1 = new MyClass();
       MyClass t2 = new MyClass();

       System.out.println(t1.x + " " + t2.x); // Imprime: 10 10
   }
}
```

En la línea:

```
      MyClass t1 = new MyClass();
```

el constructor MyClass() se invoca en el objeto t1 y asigna el valor 10 a t1.x.

Constructores con parámetros.

En un constructor los parámetros se añaden como si fuese un método; basta con declararlos entre paréntesis tras el nombre del constructor:

[ConstructorParameters.java]

```
class MyClass {
   int x;

   // El constructor de MyClass
   MyClass(int i) {
       x = i;
   }
}

class ConstructorParameters {
   public static void main(String args[]) {
       MyClass t1 = new MyClass(10);
       MyClass t2 = new MyClass(88);

       System.out.println(t1.x + " " + t2.x); // Imprime: 10 88
   }
}
```

La palabra clave this.

Al invocar un método, se le pasa automáticamente un argumento implícito que es una referencia al objeto invocador (es decir, el objeto en el que se invoca el método). Esta referencia se denomina 'this'. Para entenderla, veamos un programa que crea la clase Pwr que calcula el resultado de un número elevado a una potencia entera:

```
class Pwr {
   // Variables miembros de Pwr
   double b;
   int e;
   double val;

   // Constructor de la calse Pwr.
   Pwr(double base, int exp){
       b = base;
```

```
        e = exp;
        val = 1;
        if(exp==0) return;
        for( ; exp>0; exp--) val = val * base;
    }
    // Método devolver valor.
    double get_pwr() {
        return val;
    }
}

class WordThis {
    public static void main(String args[]) {
        Pwr x = new Pwr(4.0, 2);
        Pwr y = new Pwr(2.5, 1);
        Pwr z = new Pwr(5.7, 0);

        System.out.println(x.b + x.e + x.get_pwr());
        System.out.println(y.b + y.e + y.get_pwr());
        System.out.println(z.b + z.e + z.get_pwr());
    }
}
```

Como ya se sabe, en un método se puede acceder directamente a otros miembros de una clase, sin clasificación de objetos o clases. Por tanto, dentro de get_pwr(), la instrucción

```
return val;
```

significa que se devuelve la copia de val asociada al objeto invocador. Sin embargo, la misma instrucción se puede escribir de esta forma:

```
return this.val;
```

Aquí, this hace referencia al objeto en el que se invoca get_pwr(). Por lo tanto, this.val hace referencia a la copia de val de ese objeto. Crear una instrucción sin usar this es simplemente un abreviatura.

[WordThis.java]

```
class Pwr {
    // Variables miembros de Pwr
    double b;
    int e;
    double val;

    // Constructor de la calse Pwr.
    Pwr(double base, int exp){
        this.b = base;
        this.e = exp;
        this.val = 1;
        if(exp==0) return;
        for( ; exp>0; exp--) this.val = this.val * base;
    }
    // Método devolver valor.
    double get_pwr() {
        return this.val;
    }
}
```

```
class WordThis {
   public static void main(String args[]) {
       Pwr x = new Pwr(4.0, 2);
       Pwr y = new Pwr(2.5, 1);
       Pwr z = new Pwr(5.7, 0);

       System.out.println(x.b + x.e + x.get_pwr());
       System.out.println(y.b + y.e + y.get_pwr());
       System.out.println(z.b + z.e + z.get_pwr());

   }
}
```

En realidad, ningún programador de Java crearía Pwr de esta forma ya que no ofrece ventaja alguna y la forma estándar resulta más sencilla. Sin embargo, this tiene importantes aplicaciones. Por ejemplo la sintaxis de Java permite que el nombre de un parámetro o variable local coincida con el de una variable de instancia: Por ejemplo, el siguiente código:

```
      Pwr(double b, int e){
            this.b = b;
            this.e = e;
            val = 1;
            if(exp==0) return;
            for( ; exp>0; exp--) val = val * base;
      }
```

En esta versión, los nombres de los parámetros coinciden con los nombres de las variables de instancia, por lo que se ocultan. Se usa this para revelar las variables de instancia.

OTROS TIPOS DE DATOS Y OPERADORES

Matrices.

En Java las matrices se implementan como objetos. Para declarar una matriz unidimensional se puede usar el siguiente formato general:

```
tipo nombre-matriz[] = new tipo[tamaño];
```

Por ejemplo:

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

Esta declaración se puede dividir en dos:

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

[Array.java]

```
class Array {
    public static void main(String args[]) {
        int sample[] = new int[10];
        int i;

        for(i = 0; i < 10; i++) {
            sample[i] = i;
            System.out.println("[" + i +"] :" + sample[i]);
        }
    }
}
```

Las matrices se pueden inicializar al crearse. El formato general para inicializar una matriz unidimensional es le siguiente:

```
tipo nombre[] = {valor1, valor2, ..., valorN};
```

No es necesario usar explícitamente el operador new.

[InitializeArray.java]

```
class InitializeArray {
    public static void main(String args[]) {
        int nums[] = {99, -10, 100123, 18, 49};
        int min, max;

        min = max = nums[0];
        for(int i=1; i<5; i++){
            if(nums[i] < min) min = nums[i];
            if(nums[i] > max) max = nums[i];
        }

        System.out.println(" El mínimo y el máximo son: " + min + " " + max);
    }
}
```

Matrices de dos dimensiones.

Una matriz (array) de dos dimensiones es, en esencia, una lista de matrices unidimensionales. Para declarar una matriz de dos dimensiones con el nombre table y el tamaño 10, 20 utilice lo siguiente:

```
int table[] = new int[10][20];
```

Por ejemplo:

[TwoDimensionalArray.java]

```
class TwoDimensionalArray {
   public static void main(String args[]) {
        int t, i;
        int table[][] = new int[3][4];

        for(t=0; t < 3; t++) {
            for(i=0; i < 4; i++) {
                table[t][i] = (t*4)+i+1;
                System.out.print(table[t][i] + " ");
            }
            System.out.println();
        }
    }
}
```

Inicializar una matriz de dos dimensiones:

[InitializeTwoDimensionalArray.java]

```
class InitializeTwoDimensionalArray {
    public static void main(String args []) {
        int sqrs [][] = {
           {1, 1}, // Primera fila. Cada fila tiene sus propios inicializadores.
           {2, 4},
           {3, 9},
           {4, 16},
           {5, 25},
           {6, 36},
           {7, 49},
           {8, 64},
           {9, 81},
           {10, 100},
        };
        int i, j;

        for (i=0; i < 10; i++) {
            for(j=0; j < 2; j++)
                System.out.print(sqrs[i][j] + " ");
            System.out.println();
        }
    }
}
```

Sintaxis alternativa para declarar matrices.

Existe un segundo formato para declarar matrices:

```
tipo[] nombre-var;
```

Los corchetes aparecen tras el especificador de tipo, no del nombre de la variable matriz. Por ejemplo, las dos siguientes declaraciones son equivalentes:

```
int counter[] = new int[3];
int[] counter = new int[3];
```

Esta declaración alternativa es muy útil para declarar varias matrices al mismo tiempo:

```
int[] nums, nums2, nums3; // se declaran tres matrices simultáneamente.
```

También resulta útil para especificar una matriz como tipo devuelto por un método. Por ejemplo:

```
int[] someMeth() { ...}
```

declara que someMeth devuelve una matriz de tipo int.

Usar el miembro length.

Como las matrices se implementan como objetos, cuentan con una variable de instancia length asociada que contiene el número de elementos que la matiz puede almacenar, es decir length contiene el tamaño de la matriz. Veamos un ejemplo. El siguiente programa usa length para copiar una matriz en otra al tiempo que evita que se superen sus límites y se genere una excepción en tiempo de ejecución.

[LengthCopyArray.java]

```
class LengthCopyArray {
   public static void main(String args[]) {
       int i;
       int nums1[] = new int[10];
       int nums2[] = new int[10];

       for(i=0; i < nums1.length; i++)
           nums1[i] = i;

       // copiar nums1 en muns2
       if(nums2.length >= nums1.length) // Usar length para comparar tamaños.
           for(i=0; i < nums1.length; i++)
               nums2[i] = nums1[i];

       for(i=0; i < nums2.length; i++)
           System.out.print(nums2[i] + " ");
   }
}
```

El bucle for de estilo for-each. Bucle for-each.

Al trabajar con matrices es habitual toparse con casos en los que se requiere examinar todos los elementos de la matriz, de principio a fin. Java define una

segunda variante del bucle for par hacer esto más sencillo. Esta segunda forma del bucle for implementa un bucle estilo for-each. Itera por una colección de objetos, como una matriz, de forma secuencial, de principio a fin.

El formato general del bucle for-each es el siguiente:

```
for(tipo var-iteración: colección) bloque-de-instrucciones
```

En cada iteración del bucle se recupera el siguiente elemento de la colección y se almacena en var-iteración. El bucle se repite hasta obtener todos los elementos de la colección.

Veamos un bucle for tradicional:

```
class For {
    public static void main(String args[]){
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        for(int i = 0;i<10;i++) sum += nums[i];

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

Esta operación con un bucle for-each, quedaría como sigue:

```
class ForEach {

    public static void main(String args[]){
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        for(int x: nums) sum += x;  // Bucle for-each

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

El siguiente programa usa un bucle for, estilo for-each, para buscar un valor en una matriz sin ordenar. Se detiene al encontrar el valor.

```
class SearchForEach {
    public static void main(String args[]) {
        int nums[] = {6, 8, 3, 7, 5, 6, 1,4};
        int val = 5;
        boolean found = false;

        for(int x: nums) {
            if(x == val) {
                found = true;
                break;
            }
        }
        if(found)
            System.out.println("¡Valor encontrado!");
    }
}
```

Cadenas.

En otros lenguajes de programación, una cadena es una matriz de caracteres, pero en Java no, en Java las cadenas son objetos.

Puede crear una cadena como cualquier otro objeto, por medio de new y la invocación del constructor String:

```
String str = new String("Hello");
```

Puede crear una cadena a partir de otra:

```
String str = new String("Hello");
String str2 = new String(str);
```

Otra forma de crear una cadena es la siguiente:

```
String str = "En Java las cadenas son poderosas";
```

La clase String contiene diversos métodos para trabajar con cadenas, se muestran algunos:

```
boolean equals(cadena)  Devuelve true si la cadena de invocación es igual.
int length()            Obtiene la longitud de una cadena.
char charAt(índice)     Obtiene el carácter en el índice especificado.
int indexOf(cadena)     Busca (cadena) en la cadena, devuelve el índice.
int lastIndexOf(cadena) Busca (cadena) en la cadena, devuelve el índice.
int compareTo(cadena)   Compara las cadenas, devuelve un int.
```

Matrices de cadenas.

[StringArrays.java]

```
class StringArrays {
    public static void main(String args[]) {
        String strs[] = {"Esto", "es", "un", "test"};

        System.out.println("Matriz original: ");
        for(String s: strs)
            System.out.print(s + " ");
        System.out.println("\n");

        // Cambiar cadena.
        strs[1] = "fue";
        strs[3] = "test también";

        System.out.println("Matriz modificada: ");
        for(String s: strs)
            System.out.print(s + " ");
    }
}
```

Usar strings en switch.

[Stringswitch.java]

```
class Stringswitch {
    public static void main(String args[]) {
        String command = "cancel";
```

```
        switch (command) {
            case "connect":
                System.out.println("Conectando");
                break;
            case "cancel":
                System.out.println("Cancelando");
                break;
            case "disconnect":
                System.out.println("Desconectando");
                break;
            default:
                System.out.println("Error de comando");
                break;
        }
    }
}
```

Utilizar argumentos en la línea de comandos. Parámetro 'args' de main().

Muchos programas aceptan argumentos de línea de comandos. Resulta muy sencillo acceder a los argumentos de línea de comandos de un programa Java, ya que se almacenan como cadenas en la matriz String pasada a main().

El siguiente programa muestra todos los argumentos de línea de comandos con los que se invoca:

[CommandLineArguments.java]

```
class CommandLineArguments {
    public static void main(String args []) {
        System.out.println("Estos son " + args.length + " argumentos de cli");

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

El operador ?.

Uno de los operadores más fascinantes de Java es '?'. Puede usarse para reemplazar instrucciones if-else con este formato:

```
    if(condición)
        var = expresión1;
    else
        var = expresión2;
```

Aquí el valor asignado a var depende del resultado de la condición que controla la instrucción if. El operador ? se denomina operador ternario ya que requiere tres operandos:

```
     Exp1 ? Exp2 : Exp3;
```

donde Exp1 es una expresión booleana y Exp2 y Exp3 son expresiones de cualquier tipo menos void. El tipo de Exp2 y Exp3 debe ser el mismo o compatible.

Se evalúa Exp1. Si es true, se evalúa Exp2 y se convierte en el valor de toda la

expresión ?. Si Exp1 es false, se evalúa Exp3 y su valor se convierte en el de toda la expresión.

Veamos un ejemplo con un programa que divide dos número pero no permite la división por cero.

[TernaryOperator.java]

```
class TernaryOperator {
    public static void main(String args[]) {
        int result;

        for(int i = -5; i < 6; i++) {
            result = (i != 0) ? 100 / i : 0; // Impide la división por cero
            if(i != 0)
                System.out.println("100 / " + i + " es " + result);
        }
    }
}
```

En el ejemplo anterior, se asigna a result el resultado de la división de 100 por i. Sin embargo, esta división solo se produce si i no es cero. Cuando i es cero, se asigna un valor cero de marcador de posición a result. En realidad no es necesario asignar el valor generado por ? a una variable. Por ejemplo, puede usar el valor como argumento en la invocación de un método o, si las expresiones son todas de tipo boolean, se puede usar ? Como expresión condicional en un bucle o una instrucción if, por ejemplo:

[TernaryOperator2.java]

```
class TernaryOperator2 {
    public static void main(String args[]) {

        for(int i = -5; i < 6; i++) {
            if((i != 0) ? true : false)
                System.out.println("100 / " + i + " es " + 100 / i);
        }
    }
}
```

ANÁLISIS DETALLADO DE MÉTODOS Y CLASES

Controlar el acceso a los miembros de una clase.

Los miembros de una clase son un conjunto de elementos que definen a los objetos (atributos o propiedades), así como los comportamientos o funciones (métodos) que maneja el objeto.

Una clase proporciona el medio para poder controlar el acceso a miembros. Hay dos tipos básicos de miembros de clase: públicos y privados. Un miembro público es accesible para el código definido fuera de su clase. Es el tipo de miembro que hemos usado hasta ahora. Un miembro privado, en cambio, solo es accesible por otros métodos definidos por su clase. El acceso se controla a través de miembros privados.

La restricción del acceso a los miembros de una clase es una parte fundamental de la programación orientada a objetos, ya que evita el uso incorrecto del objeto. Al permitir el acceso a datos privados solamente a través de un conjunto bien definido de métodos, puede impedir que se asignen valores inadecuados a dichos datos. El código externo de la clase no puede establecer directamente el valor de un miembro privado. También puede controlar cómo y cuándo se usan los datos dentro de un objeto. Por tanto, si se implementa correctamente, una clase crea una caja negra que se puede usar, pero sin acceso a su funcionamiento interno.

Modificadores de acceso.

El control de acceso a miembros se consigue a través de tres modificadores de acceso: public, private y protected. Si no se usa un modificador de acceso se asume la configuración predeterminada, por lo general, pública. El modificador protected solo se aplica en casos de herencia.

Al modificar un miembro de una clase con el especificador public, cualquier código del programa puede acceder a ese miembro, incluidos los métodos definidos en otras clases.

Al especificar un miembro de una clase como private, solo puede acceder al mismo otros miembros de su clase. Por los tanto los métodos de otras clases no pueden acceder a un miembro private de otra clase.

La configuración de acceso predeterminada (cuando no se usan modificadores de acceso) es igual que public a menos que el programa se divida en paquetes. Un paquete es una agrupación de clases.

Para comprender los efectos de public y private veamos el siguiente programa:

[AccessDemo.java]

```
//Acceso público frente a privado.

class MyClass {
    private int alpha; // acceso privado.
    public int beta;   // acceso público.
    int gamma; // acceso predeterminado (básicamente es público).

    /* Métodos para acceder a alpha. Un miembro de uan clase puede acceder a un
    miembro privado de la misma clase
    */
    void setAlpha(int a) {
```

```
        alpha = a;
    }

    int getAlpha() {
        return alpha;
    }
}

class AccessDemo {
    public static void main(String args[]) {
        MyClass ob = new MyClass();

        /* El acceso a alpha solo se permite a través de sus
        métodos de acceso.
        */
        ob.setAlpha(-99);
        System.out.println("ob.alpha es " + ob.getAlpha());

        /* No se puede acceder a alpha de esta forma: */
        //ob.alpha = 10; // ¡Error! ¡alpha es privated!

        /* Correcto ya que beta y gamma son public. */
        ob.beta = 88;
        ob.gamma = 99;
    }
}
```

La clave es que un miembro o método privado puede usarse libremente en otros miembros de su clase pero no es accesible a código externo a su clase.

Pasar objetos a métodos.

[PassObject.java]

```
// Se pueden pasar objetos a métodos.
class Block {
    int a, b, c;
    int volume;

    Block(int i, int j, int k){ // Constructor
        a = i;
        b = j;
        c = k;
        volume = a* b * c;
    }

    // Devolver true si ob define el mismo bloque (Block).
    boolean sameBlock(Block ob) {   // Usar tipo de objeto como parámetro.
        if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
        else return false;
    }

    // Devolver true si ob define el mismo valor (volume).
    boolean sameVolume(Block ob) {   // Usar tipo de objeto como parámetro.
        if(ob.volume == volume) return true;
        else return false;
    }
}
```

```
class PassObject {
    public static void main(String args[]) {
        Block ob1 = new Block(10, 2, 5);
        Block ob2 = new Block(10, 2, 5);
        Block ob3 = new Block(4, 5, 5);

        System.out.println("ob1 = dimensiones que ob2: " + ob1.sameBlock(ob2));
        // Pasar un obejto.
        System.out.println("ob1 = dimensiones que ob3: " + ob1.sameBlock(ob3));
        System.out.println("ob1 = volumen que ob3: " + ob1.sameVolume(ob3));
    }
}
```

Cómo pasar argumentos. Paso por valor y paso por referencia.

En determinados casos, los efectos de pasar un objeto aun método serán diferentes de los experimentados al pasar argumentos que no son objetos.

Existen dos formas en la que un argumento se puede pasar a una subrutina: invocación por valor e invocación por referencia. En el paso por valor se copia el valor de un argumento en el parámetro formal de la subrutina. Por tanto, los cambios realizados en el parámetro de la rutina no afectan al argumento de la invocación. En el paso por referencia, se pasa una referencia a un argumento (no el valor) al parámetro. En la subrutina, esta referencia se usa para acceder al argumento real especificado en la invocación. Esto significa que los cambios realizados en el parámetro si afectan al argumento utilizado para invocar a la subrutina.

Al pasar un tipo primitivo, como int o double, a un método, se pasa por valor. Por tanto, se crea una copia del argumento y lo que sucede al parámetro que recibe el argumento no afecta al exterior del método.

Al pasar un objeto a un método, la situación cambia. Los objetos se pasan implícitamente por referencia. Recuerde que al crear una variable de un tipo de clase, se crea una referencia a un objeto. Es la referencia, no el objeto, la que se pasa al método. Como resultado, al pasar esta referencia a un método, el parámetro que lo recibe hace referencia al mismo objeto al que hace referencia el argumento. Por tanto, los objetos se pasan a métodos por invocación de referencia. Los cambios realizados en el objeto dentro del método afectan al objeto usado como argumento.

Ejemplo de paso por valor de tipos primitivos.

```
// Los tipos primitivos se pasan por valor.
class Test {
    /* Este método no cambia los argumentos usados en la invocación.*/
    void noChange(int i, int j) {
        i = i + j;
        j = -j;
    }
}

class PassByValue {
    public static void main(String args[]) {
        Test ob = new Test();

        int a = 15, b = 20;
        // a y b antes de la llamada: 15 20
        System.out.println("a y b antes de la llamada: " + a + " " + b);
```

```
        ob.noChange(a, b);
        // a y b después de la llamada: 15 20
        System.out.println("a y b después de la llamada: " + a + " " + b);
    }
}
```

Como puede apreciarse, las operaciones dentro de noChange() no afectan a los valores de a y b usados en la invocación.

Ejemplo de paso por referencia de un objeto.

```
// Los objetos se pasan a través de sus referencias.
class Test {
    int a,b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    /* Pasar un objeto. ob.a y ob.b del objeto usado en la invocación
    cambian.*/
    void change(Test ob) {
        ob.a = ob.a + ob.b;
        ob.b = -ob.b;
    }
}

class PassByRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);

        System.out.println("ob.a y ob.b antes de la llamada: "
        + ob.a + " " + ob.b); // ob.a y ob.b antes de la llamada: 15 20

        ob.change(ob);

        System.out.println("ob.a y ob.b después de la llamada: "
        + ob.a + " " + ob.b); // ob.a y ob.b después de la llamada: 35 -20
    }
}
```

En este caso, las acciones dentro de change() afectan al objeto usado como argumento.

Devolver objetos.

[ReturnStringObject.java]

```
// Devolver un objeto String.
class ErrorMsg {
    String msgs[] = {
        "Output Error",
        "Input error",
        "Disk Full",
        "Index Out-Of_bounds"
    };

    // Devolver el mensaje de error.
    String getErrorMsg(int i) { // Devolver un objeto tipo String.
```

```
        if(i >= 0 & i < msgs.length)
            return msgs[i];
        else
            return "Invalid Error Code";
    }
}

class ReturnStringObject {
    public static void main(String args[]) {
        ErrorMsg err = new ErrorMsg();
        System.out.println(err.getErrorMsg(2)); // Disk Full
        System.out.println(err.getErrorMsg(19));    // Invalid Error Code
    }
}
```

[ReturnObject.java]

```
// Devolver un objeto definido por el programador.
class Err {
    String msg; // mensaje de error.
    int severity; // código de la gravedad del error.

    Err(String m, int s) {
        msg = m;
        severity = s;
    }
}

class ErrorInfo {
    String msgs[] = {
        "Output Error",
        "Input error",
        "Disk Full",
        "Index Out-Of_bounds"
    };

    int howbad[] = {3, 3, 2, 4};

    Err getErrorInfo(int i) { // Devolver un objeto tipo Err.
        if(i >= 0 & i < msgs.length)
            return new Err(msgs[i], howbad[i]);
        else
            return new Err("Inalid Error Code", 0);
    }
}

class ReturnObject {
    public static void main(String args[]) {
        ErrorInfo err = new ErrorInfo();
        Err e;

        e = err.getErrorInfo(2);
        // Disk Full severity: 2
        System.out.println(e.msg + " severity: " + e.severity);

        e = err.getErrorInfo(19);
        // Inalid Error Code severity: 0
        System.out.println(e.msg + " severity: " + e.severity);
    }
}
```

Sobrecargar métodos.

En el ámbito de la POO, la sobrecarga de métodos se refiere a la posibilidad de tener dos o más métodos de la misma clase con el mismo nombre pero distinta funcionalidad. Es decir, dos o más métodos con el mismo nombre realizan acciones diferentes y el compilador usará una u otra dependiendo de los parámetros usados.

Como se entiende de lo anterior, las declaraciones de parámetros de los métodos deben ser diferentes.

La sobrecarga de métodos es una de las técnicas de Java para implementar el polimorfismo.

Al invocar un método sobrecargado, se ejecuta la versión del método cuyos parámetros coincidan con los argumentos.

El tipo y/o el número de parámetros de cada método sobrecargado debe ser diferente. No basta con que dos métodos difieran solo en los tipos devueltos, que no ofrecen información suficiente para que Java decida qué método usar.

La sobrecarga es un proceso transparente para el programador del que se encarga el compilador.

[OverloadDemo.java]

```java
// Sobrecarga de métodos.

class Overload {
    void ovlDemo() { // Primera versión.
        System.out.println("Sin parámetros");
    }

    // Sobrecargar ovlDemo para un parámetro entero.
    void ovlDemo(int a) { // Segunda versión.
        System.out.println("Un parámetro: " + a);
    }

    // Sobrecargar ovlDemo para dos parámetros enteros.
    int ovlDemo(int a, int b) {  // Tercera versión.
        System.out.println("Dos parámetros: " + a + " " + b);
        return a + b;
    }

    // Sobrecargar ovlDemo para dos parámetros double.
    double ovlDemo(double a, double b) {  // Cuarta versión.
        System.out.println("Dos parámetros: " + a + " " + b);
        return a + b;
    }

}

class OverloadDemo {
    public static void main(String args[]) {
        Overload ob = new Overload();
        int resI;
        double resD;

        // invocar todas las versiones de ovlDemo()
        ob.ovlDemo();
```

```
        System.out.println();

        ob.ovlDemo(2);
        System.out.println();

        resI = ob.ovlDemo(4, 6);
        System.out.println(resI);

        resD = ob.ovlDemo(1.1, 2.32);
        System.out.println(resD);
    }
}
```

La importancia de la sobrecarga de métodos. IMPORTANTE: en el lenguaje C, por ejemplo, que no soporta la sobrecarga, la función abs() devuelve el valor absoluto de un entero, labs() el valor absoluto de un entero long y fabs() devuelve el valor absoluto de un valor de coma flotante. Cada función necesita un nombre propio, aunque las tres funciones hagan lo mismo. La biblioteca de clases estándar de Java incluye un método de valor absoluto, abs(), que la clase Math de Java sobrecarga para trabajar con todos los tipos numéricos. Java determina que versión de abs() invocar en función del tipo de argumento.

¿Qué es una signatura o firma en Java?.

Una firma es el nombre de un método más su lista de parámetros; por tanto dos métodos de la misma clase no pueden tener la misma firma. La firma no incluye el tipo devuelto (en algunos lenguajes sí), ya que Java no lo utiliza para la resolución de sobrecarga.

Sobrecarga de constructores.

Al igual que los métodos, los constructores también pueden sobrecargarse. De este modo puede crear objetos de distintas formas. En el ejemplo MyClass() se sobrecarga de cuatro formas diferentes y en cada una de ellas crea un objeto distinto. Se invoca el constructor adecuado en función de los parámetros especificados al ejecutar new. Al sobrecargar el constructor de una clase el usuario de la misma dispone de gran flexibilidad para crear objetos.

Uno de los motivos más habituales de la sobrecarga de constructores es para que un objeto pueda inicializar otro.

[OverloadConstructor.java]

```
// Ejemplo de sobrecarga de constructor.
class MyClass {
    int x;

    MyClass() { // Crear objetos de distintas formas.
        System.out.println("Inside MyClass().");
        x = 0;
    }
    MyClass(int i) { // Crear objetos de distintas formas.
        System.out.println("Inside MyClass(int).");
        x = i;
    }
    MyClass(double d) { // Crear objetos de distintas formas.
        System.out.println("Inside MyClass(double).");
```

```
        x = (int) d;
    }
    MyClass(int i, int j) { // Crear objetos de distintas formas.
        System.out.println("Inside MyClass(int, int).");
        x = i * j;
    }
}

class OverloadConstructor {
    public static void main(String args[]) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass(88);
        MyClass t3 = new MyClass(17.23);
        MyClass t4 = new MyClass(2, 4);

        System.out.println("t1.x " + t1.x);
        System.out.println("t2.x " + t2.x);
        System.out.println("t3.x " + t3.x);
        System.out.println("t4.x " + t4.x);
    }
}
```

Static.

En ocasiones tendrá que definir un miembro de clase que se use independientemente de los objetos de esa clase. Por lo general se accede a un miembro de clase a través de un objeto de su clase, pero se puede crear un miembro que usar de forma independiente, sin referencias a una instancia concreta. Para crear este tipo de miembro, debe preceder su declaración con la palabra clave 'static'. Al declarar un miembro como static, se puede acceder al mismo antes de crear objetos de su clase y sin referencias a ningún objeto.

Puede declarar como static tanto métodos como variables. El ejemplo más habitual de miembro static es main(), ya que debe invocarse por la MVJ al iniciar el programa. Fuera de la clase, para usar un miembro static, basta con especificar el nombre de su clase seguido del operador '.'. No es necesario crear objetos.

Por ejemplo si se desea asignar el valor 10 a una variable 'static count' que forma parte de la clase Timer, se usa esta línea:

```
Timer.count = 10;
```

Es un formato similar al utilizado para acceder a variables de instancia normales a través de un objeto, con la excepción de que se usa el nombre de la clase.

Un método static se puede invocar de la misma forma, por medio del operador punto en el nombre de la clase.

Las variables declaradas como static son, básicamente, variables globales. Al declarar un objeto, no se crean copias de una variable static. En su lugar, todas las instancias de la clase comparten la misma variable static.

Veamos un ejemplo que ilustra las diferencias entre una variable static y una variable de instancia:

[StaticMember.java]

```
// Uso de una varible stática.
class StaticDemo {
    int x;  // variable de instancia normal.
    static int y;   // variable estática.

    // Dovolver la suma.
    int sum() {
        return x + y;
    }
}
class StaticMember {
    public static void main(String args[]) {
        StaticDemo ob1 = new StaticDemo();
        StaticDemo ob2 = new StaticDemo();

        // Cada objeto tiene su copia de una variable de instancia.
        ob1.x = 10;
        ob2.x = 20;
        System.out.println("Por supuesto, ob1.x y ob2.x son independientes");
        System.out.println("ob1.x: " + ob1.x + "\nob2.x: " + ob2.x);
        System.out.println();

        // Cada objeto comparte una copia de la variable estática.
        StaticDemo.y = 19;
        System.out.println("La variable estática y es comaprtida");
        System.out.println("Asignado el valor 19 a StaticDemo.y");

        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println();

        StaticDemo.y = 100;
        System.out.println("Cambiada StaticDemo.y a 100");

        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println();
    }
}
```

Como puede comprobar, la variable static y se comparte en ob1 y ob2. Al cambiarla, toda la clase se ve afectada, no solo una instancia.

La diferencia entre un método static y uno normal es que el método static se invoca a través de su nombre de clase, sin crear objetos de dicha clase. Veamos un ejemplo que crea un método static.

[StaticMethod.java]

```
// Usar un método estático.
class SDemo {
    static int val = 1024;

    // un método estático.
    static int valDiv2() {
        return val/2;
    }
}
```

```
class StaticMethod {
    public static void main(String args[]) {
        System.out.println("val es " + SDemo.val);
        System.out.println("SDemo.valDiv2(): " + SDemo.valDiv2());

        SDemo.val = 4;

        System.out.println("val es " + SDemo.val);
        System.out.println("SDemo.valDiv2(): " + SDemo.valDiv2());
    }
}
```

Los métodos declarados como static tienen ciertas restricciones:

- Solo pueden invocar directamente otros métodos static.
- Solo pueden acceder directamente a datos static.
- Carecen de una referencia this.

No se puede acceder a un variable normal (no estática) desde un método estático.

Bloques estáticos.

Un bloque estático es un bloque que se ejecuta al cargar la clase por primera vez. Por ejemplo, se puede necesitar establecer una conexión a un sitio remoto o inicializar determinadas variables static antes de poder usar los métodos static de la clase. Veamos un ejemplo de un bloque static.

[StaticBlock.java]

```
// Usar un bloque estático.
class SBlock {
    static double root0f2;
    static double root0f3;

    static { // Bloque static. Este bloque se ejecuta al cargar la clase.
        System.out.println("Dentro de un bloque estático");
        root0f2 = Math.sqrt(2.0);
        root0f3 = Math.sqrt(3.0);
    }

    SBlock(String msg) {
        System.out.println(msg);
    }
}

class StaticBlock {
    public static void main(String args[]) {
        SBlock ob = new SBlock("Dentro del constructor");

        System.out.println("Raíz cuadrada de 2 es: " + SBlock.root0f2);
        System.out.println("Raíz cuadrada de 3 es: " + SBlock.root0f3);

    }
}
```

El ejemplo anterior imprime:

```
Dentro de un bloque estático
```

```
Dentro del constructor
Raíz cuadrada de 2 es: 1.4142135623730951
Raíz cuadrada de 3 es: 1.7320508075688772
```

Como se puede comprobar, el bloque static se ejecuta antes de crear ningún objeto.

varargs: argumentos de longitud variable.

Un método que acepta un número variable e argumentos se denomina método varargs. La lista de parámetros de un método varargs no es fija, sino de longitud variable. Por tanto, un método varargs puede aceptar un número variable de argumentos.

Supongamos un método que abra una conexión a Internet, puede aceptar el nombre de un usuario, la contraseña, el nombre de archivo, el protocolo y demás, pero proporcionar valores predeterminados si parte de esa información no se proporciona.

Un argumento de longitud variable se especifica por medio de tres puntos [...]. el siguiente ejemplo crea el método vaTest(), que especifica un número variable de argumentos:

```
class VaTest {
    //vaTest() usa un número variable de argumentos.
    static void vaTest(int ... v) {
        System.out.println("Número de argumentos: " + v.length);
        System.out.println("Contiene: ");

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

Esta sintaxis indica al compilador que vaTest() se puede invocar con cero o más argumentos. Es más, hace que v se declare implícitamente como matriz de tipo int[]. Por tanto, dentro de vaTest(), se accede a v por medio de la sintaxis estándar para parámetros.

[VarArgs.java]

```
// Ejemplo de argumentos de longitud variable.
class VarArgs {
    // vaTest usa un argumento de longitud variable.
    static void vaTest(int ... v) {
        System.out.println("Número de argumentos: " + v.length);
        System.out.println("Contiene: ");

        for(int i = 0; i < v.length; i++)
            System.out.println("Argumento " + i + ": " + v[i]);
        System.out.println();
    }

    public static void main(String args[]) {
        //Se puede invocar vaTest() con un número variable de argumentos.
        vaTest(10); // 1 argumento
        vaTest(1, 2, 3); // 3 argumentos
        vaTest(); // Sin argumentos
    }
}
```

El programa anterior imprime lo siguiente:

```
Número de argumentos: 1
Contiene:
Argumento 0: 10

Número de argumentos: 3
Contiene:
Argumento 0: 1
Argumento 1: 2
Argumento 2: 3

Número de argumentos: 0
Contiene:
```

Un método puede tener argumentos normales, además de un parámetro de longitud variable. Sin embargo, el parámetro de longitud variable debe ser el último declarado por el método. Por otro lado, solo puede haber un parámetro varargs. Por ejemplo:

```
int doIt(int a, int b, double c, int ... vals) {
```

Sobrecarga de métodos varargs.

[OverloadVarArgs.java]

```
class OverloadVarArgs {

    static void vaTest(int ... v) {
        System.out.println("Número de argumentos vaTest(int ...): " + v.length);
        System.out.println("Contiene: ");

        for(int i = 0; i < v.length; i++)
            System.out.println("Argumento " + i + ": " + v[i]);
        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.println("Número de argumentos vaTest(boolean ...): "
                           + v.length);
        System.out.println("Contiene: ");

        for(int i = 0; i < v.length; i++)
            System.out.println("Argumento " + i + ": " + v[i]);
        System.out.println();
    }

    static void vaTest(String msg, int ... v) {
        System.out.println("Número de argumentos vaTest(String msg, int ...): "
                          + v.length);
        System.out.println("Contiene: ");

        for(int i = 0; i < v.length; i++)
            System.out.println("Argumento " + i + ": " + v[i]);
        System.out.println();
    }

    public static void main(String args[]) {
        vaTest(1, 2, 3);
```

```
        vaTest("Testing: ", 10, 20);
        vaTest(true, false, false);
    }
}
```

El programa anterior imprime:

```
Número de argumentos vaTest(int ...): 3
Contiene:
Argumento 0: 1
Argumento 1: 2
Argumento 2: 3

Número de argumentos vaTest(String msg, int ...): Testing: 2
Contiene:
Argumento 0: 10
Argumento 1: 20

Número de argumentos vaTest(boolean ...): 3
Contiene:
Argumento 0: true
Argumento 1: false
Argumento 2: false
```

HERENCIA

La herencia es una de los tres principios fundamentales de la programación orientada a objetos. Por medio de la herencia, puede crear una clase general, que defina rasgos comunes, a una serie de elementos relacionados. Esta clase se puede heredar por otras y cada una de estas añadir elementos más concretos.

En Java, una clase heredada se denomina superclase y la clase que realiza la herencia se denomina subclase. Es decir, una subclase es una versión especializada de una superclase. Hereda todas las variables y métodos definidos por la superclase y añade los suyos propios.

Aspectos básicos de la herencia.

Java admite la herencia al permitir que una clase incorpore otra en su declaración. Parra ello utiliza la palabra reservada 'extends'. La subclase añade o amplia la superclase.

[Subclase.java]

```
// Ejemplo de superclase y subclase.

// Una clase para objetos de dos dimensiones.
class TwoDShape {
    double width;
    double height;

    void showDim() {
        System.out.println("Ancho y alto son " + width + " y " + height);
    }
}

// Una subclase de TwoDShape para triángulos.
class Triangle extends TwoDShape { // Triangle se hereda de TwoDShape.
    String style;

    double area() {
        return width * height / 2; /* Triangle puede hacer referencia a los
                                      miembro de TwoDShape como si fuesen parte de
                                      Triangle */
    }

    void showStyle() {
        System.out.println("El triángulo es " + style);
    }
}

class Subclase {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        /* Todos los miembros de Triangle están disponibles para objetos
          Triangle, incluso los heredados de TwoDShape.
        */
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "Relleno";
```

```
        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "Delineado";

        System.out.println("Información para t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("El área es: " + t1.area());

        System.out.println();

        System.out.println("Información para t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("El área es: " + t2.area());
    }
}
```

Aunque TwoDShape es una superclase de Triangle, también es una clase independiente. El ser superclase de una subclase no significa que no se pueda usar por sí sola. Evidentemente un objeto de TwoDShape no conoce ni puede acceder a los miembros de Triangle.

Solo puede especificar una superclase por subclase, Java no admite la herencia múltiple, como sí lo hace C++. Se puede crear una jerarquía de clases en la que una subclase se convierta en la superclase de otra.

Una de las ventajas de la herencia es que tras crear una superclase que defina los atributos comunes a un grupo de objetos, se puede usar para crear todas las subclases específicas que desee.

Acceso a miembros y herencia.

Como vimos, una variable puede declararse como private para evitar su uso no autorizado. IMPORTANTE: Por tanto, aunque una subclase incluya todos los miembros de su superclase, no puede acceder a los que se hayan declarado como private. Por ejemplo aquí:

```
/* Los miembros privados (private) no se heredan.
   Este ejemplo no se compila. */
class TwoDShape {
    private double width;
    private double height;

    void showDim() {
        System.out.println("Ancho y alto son " + width + " y " + height);
    }
}
class Triangle extends TwoDShape { // Triangle se hereda de TwoDShape.
    String style;

    double area() {
        //return width * height / 2; // ¡Error! No se puede acceder.

    }

    void showStyle() {
        System.out.println("El triángulo es " + style);
    }
}
```

La clase Triangle no se compila ya que la referencia a width y height dentro del método área() supone una violación de acceso. Como estas variables se han declarado como private, solo tienen acceso los miembros de su propia clase. Las subclases no tienen acceso a ellos. Normalmente, se usan métodos para acceder a las variables de instancia privadas.

¿Cuándo debe ser privada una variable de instancia?

No existen reglas concretas para responder esta pregunta, pero por norma general, si una variable de instancia solo va a utilizarse con métodos definidos en su clase, debe convertirse en privada. Si una variable de instancia debe estar comprendida entre ciertos límites, debe ser privada y estar únicamente disponible para los métodos de acceso.

Constructores y herencia.

Puede haber constructores de una superclase y a la vez la subclase tener también sus propios constructores. Veamos un ejemplo de constructor en subclase:

[ConstructorsAndInheritance.java]

```
// Añadir un constructor a la subclase Triangle.

// Una clase para objetos de dos dimensiones.
class TwoDShape {
    private double width;   // Ahora estos miembros son privados.
    private double height;

    // Métodos de acceso para width y height, dado que son datos privados.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ancho y alto son: " + width  + " y " + height);
    }
}

// Una subclase de TwoDShape para triángulos.
class Triangle extends TwoDShape {
    private String style;

    //Constructor
    Triangle(String s, double w, double h) {
        setWidth(w); // Inicializar la parte TwoDShape del objeto.
        setHeight(h);
        style = s;
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("El triángulo es " + style);
    }
}
```

```
class ConstructorsAndInheritance {
    public static void main(String args[]) {
        Triangle t1 = new Triangle("Relleno", 4.0, 4.0);
        Triangle t2 = new Triangle("Deliniado", 8.0, 12.0);

        System.out.println("Información para t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("El área es: " + t1.area());

        System.out.println();

        System.out.println("Información para t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("El área es: " + t2.area());
    }
}
```

El constructor de Triangle inicializa los miembros que hereda de TwoDShape junto al propio campo style.

Cuando tanto la superclase como la subclase definen constructores, el proceso es más complicado ya que es necesario ejecutar los dos constructores. En este caso se debe usar otra palabra reservada de Java: 'super', que tiene dos formatos. El primero invoca un constructor de superclase. El segundo se usa para acceder a un miembro de la superclase oculto por un miembro de una subclase.

Utilizar 'super' para invocar constructores de superclase.

Una subclase puede invocar un constructor definido por su superclase por medio de 'super':

```
super(lista-de-parámetros);
```

lista-de-parámetros especifica los parámetros que necesita el constructor de la superclase. IMPORTANTE: super() siempre debe ser la primera instrucción ejecutada dentro de un constructor de subclase.

Para ver el uso de super(), el siguiente ejemplo define un constructor en la superclase que inicializa los miembros usados más tarde en la subclase.

[Super.java]

```
// Añadir un constructor a la superclase.
class TwoDShape {
    private double width;
    private double height;

    // Constructor (en superclase) con parámetros.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Métodos de acceso para width y height, dado que son datos privados.
    double getWidth() { return width; }
```

```
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ancho y alto son: " + width  + " y " + height);
    }
}

// Una subclase de TwoDShape para triángulos.
class Triangle extends TwoDShape {
    private String style;

    //Constructor
    Triangle(String s, double w, double h) {
        super(w, h); // Invocar constructor de la superclase
        style = s;
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("El triángulo es " + style);
    }
}

class Super {
    public static void main(String args[]) {
        Triangle t1 = new Triangle("Relleno", 4.0, 4.0);
        Triangle t2 = new Triangle("Deliniado", 8.0, 12.0);

        System.out.println("Información para t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("El área es: " + t1.area());

        System.out.println();

        System.out.println("Información para t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("El área es: " + t2.area());
    }
}
```

Triangle() invoca super() con los parámetros w y h. Esto provoca la invocación del constructor TwoDShape(), que inicializa width y height con estos valores. Triangle ya no los inicializa. Solo tiene que inicializar su valor exclusivo: style.

super() puede invocar cualquier constructor definido por la superclase. El constructor ejecutad será el que coincida con los argumentos.

Repasando los conceptos (IMPORTANTE): Cuando una subclase invoca super(), invoca el constructor de su superclase inmediata. Por tanto, super() siempre hace referencia a la superclase inmediatamente por encima de la clase invocadora, incluso en una jerarquía multinivel. Además, super() siempre debe ser la primera instrucción ejecutada dentro del constructor de una subclase.

Utilizar super para acceder a miembros de una superclase.

Existe una segunda forma de usar 'super'. Actúa como 'this', pero siempre hace referencia a la superclase de la subclase en la que se usa. Su formato general es el siguiente:

```
super.miembro
```

miembro puede ser un método o una variable de instancia.

Esta forma de super suele usarse cuando los nombres de miembros de una subclase ocultan miembros con el mismo nombre en la superclase. Veamos una sencilla jerarquía de clases:

[SuperMember.java]

```
// Usar super para evitar nombres ocultos.
class A {
    int i;
}

// Crear una subclase ampliando la clase A.
class B extends A {
    int i; // Esta i oculta la i de A.

    // Constructor de B.
    B(int a, int b) {
        super.i = a; // i en A.
        i = b; // i en B.
    }

    void show() {
        System.out.println("i en superclase: " + super.i);
        System.out.println("i en sublcase: " + i);
    }
}

class SuperMember {
    public static void main(String args[]) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}
```

Aunque la variable de instancia i de B oculta i en A, super permite el acceso a la i definida en la superclase. super también se puede usar para invocar métodos ocultos por una subclase.

Jerarquía multinivel - Clases que heredan de otras -.

```
//Una jerarquía multinivel.
class TwoDShape {
    private double width;
    private double height;

    // Un constructor predeterminado.
```

```
    TwoDShape() {
        width = height = 0.0;
    }

    // Constructor con parámetros.
    TwoDShape(double w , double h) {
        width = w;
        height = h;
    }

    // Constructor para crear objeto con la misma altura y anchura.
    TwoDShape(double x) {
        width = height = x;
    }

        // Métodos de acceso para width y height.
        double getWidth() { return width; }
        double getHeight() { return height; }
        void setWidth(double w) { width = w; }
        void setHeight(double h) { height = h; }

        void showDim() {
            System.out.println("Ancho y alto son: " + width + " y " + height);
        }
}

// Ampliar TwoDShape (Creación de subclase)
class Triangle extends TwoDShape {
    private String style;

    // Un constructor predeterminado.
    Triangle() {
        super();
        style = "none";
    }

    Triangle(String s, double w, double h) {
        super(w, h); // Invocar constructor de la superclase.
        style = s;
    }

    // Constructor con un argumento.
    Triangle(double x) {
        super(x); // Invocar constructor de la superclase.
        style = "Relleno";
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("El triángulo es " + style);
    }
}

// Ampliar Triangle (incluye todos los miembros de Triangle y TwoDShape).
class ColorTriangle extends Triangle {
    private String color;

    // Constructor
```

```
    ColorTriangle(String c, String s, double w, double h) {
        super(s, w, h); // Invoca al constructor de Triangle.
        color = c;
    }

    String getColor() { return color; }
    void showColor() {
        System.out.println("El color es " + color);
    }
}

class MultilevelClasses {
    public static void main(String args[]) {
        ColorTriangle t1 = new ColorTriangle("Azul", "Delineado", 8.0, 12.0);
        ColorTriangle t2 = new ColorTriangle("Rojo", "Relleno", 2.0, 2.0);

        System.out.println("Información para t1: ");
        /* Un objeto ColorTriangle puede invocar invocar lso métodos que defina
         y los de su superclase */
        t1.showStyle();
        t1.showDim();
        t1.showColor();
        System.out.println("El área es " + t1.area());

        System.out.println("Información para t2: ");

        t2.showStyle();
        t2.showDim();
        t2.showColor();
        System.out.println("El área es " + t2.area());
    }
}
```

El ejemplo anterior ilustra un aspecto importante: super() siempre hace referencia al constructor de la superclase más próxima. En ColorTriangle, super() invoca el constructor de Triangle. En Triangle, super() invoca el constructor de TWoDShape. En una jerarquía de clases, si el constructor de una superclase requiere parámetros, todas las subclases deben pasar dichos parámetros en el camino, independientemente de que la subclase necesite sus propios parámetros.

¿En qué orden se ejecutan los constructores?

Cuando se crea un objeto de una subclase, ¿qué constructor se ejecuta primero, el se su clase o el de la superclase?. Por ejemplo en una subclase B y una superclase A, ¿se invoca el constructor de A antes que el de B o al contrario? Los constructores se invocan en orden de derivación, de superclase a subclase. Veamos un ejemplo:

[OrderOfConstruction.java]

```
// Ejemplo de invocación de constructores.

// Crear una superclase.
class A {
    A() {
        System.out.println("Constructor de A");
    }
}
```

```
class B extends A {
    B() {
        System.out.println("Constructor de B");
    }
}

class C extends B {
    C() {
        System.out.println("Constructor de C");
    }
}

class OrderOfConstruction {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

El ejemplo anterior tiene como salida:

```
Constructor de A
Constructor de B
Constructor de C
```

Como se puede apreciar los constructores se invocan en orden de derivación.

Reemplazar métodos.

En una jerarquía de clases, cuando un método de una subclase tiene el mismo tipo de devolución y firma que un método de su superclase, el método de la subclase reemplaza al de la superclase. Al invocar un método reemplazado desde una subclase, siempre hace referencia a la versión del método definida por la subclase. La versión del método definida por la superclase se oculta.

Si se desea acceder a la versión del método de la superclase reemplazado, puede usar super.

El reemplazo de métodos solo se produce cuando las firmas de los dos métodos son idénticas. Si no lo son, los dos métodos se sobrecargan.

El reemplazo de métodos es la base de uno de los conceptos más potentes de Java: la entrega dinámica de métodos. Es el mecanismo mediante el cual una invocación a un método reemplazado se resuelve en tiempo de ejecución y no durante la compilación. La entrega dinámica de métodos es importante por ser la forma de implementar el polimorfismo en Java.

El polimorfismo es esencial par ala programación orientada a objetos ya que permite que una clase general pueda especificar métodos que sean comunes a todas sus derivadas, al tiempo que las subclases pueden definir la implementación específica de algunos o todos estos métodos.

Utilizar clases abstractas.

En ocasiones necesitará crear una superclase que solamente defina una forma generalizada que compartir en todas sus subclases para que cada una complete los detalles.

Un método abstracto se crea especificando el modificador de tipo 'abstract'.

Carece de cuerpo y, por tanto, la superclase no lo implementa. Por ello, debe reemplazarlo una subclase, no puede usar la versión definida en la superclase. Para declarar un método abstracto se usa el siguiente formato:

```
abstract tipo nombre(lista-de-parámetros);
```

El modificador abstract solo se puede usar en métodos normales, no se puede usar en métodos estáticos ni en constructores.

Una clase que contenga uno o varios métodos abstractos también debe declararse como abstracta, añadiendo el modificador abstract por delante de su declaración class. Por ello, al intentar crear un objeto de una clase abstract por medio de new se genera un error en tiempo de compilación.

Cuando una subclase hereda de una clase abstract, debe implementar todos los métodos abstractos de la superclase. En caso contrario, la subclase también debe especificarse como abstract. Por tanto, el atributo abstract se hereda hasta que se logre una implementación completa.

Veamos un ejemplo de métodos y clases abstract:

[Abstract.java]

```
// Crear una clase abstracta.
abstract class TwoDShape { // Ahora TwoDShape es abstracta.
    private double width;
    private double height;
    private String name;

    // Constructor predeterminado.
    TwoDShape() {
        width = height = 0.0;
        name = "none";
    }

    // Constructor con parámetros.
    TwoDShape(double w, double h, String n) {
        width = w;
        height = h;
        name = n;
    }

    // Crear objeto de la misma altura y anchura.
    TwoDShape(double x, String n) {
        width = height = x;
        name = n;
    }

    // Crear objeto a partir de otro.
    TwoDShape(TwoDShape ob) {
        width = ob.width;
        height = ob.height;
        name = ob.name;
    }

    // Métodos de acceso para width y height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
```

```
    String getName() { return name; }

    void showDim() {
        System.out.println("Ancho y alto son " +width + " y " + height);
    }

    // Ahora, area() es abstracto [*].
    abstract double area();
}

// Una subclase de TwoDShape para triángulos.
class Triangle extends TwoDShape {
    private String style;

    // Constructor predeterminado.
    Triangle() {
        super();
        style = "none";
    }

    // Constructor para Triangle.
    Triangle(String s, double w, double h) {
        super(w, h, "Triángulo");
        style = s;
    }

    // Constructor de un argumento.
    Triangle(double x) {
        super(x, "Triángulo"); // Invocar constructor de la superclase.
        style = "Relleno";
    }

    // Crear un objeto a partir de otro.
    Triangle(Triangle ob) {
        super(ob); // Pasar objeto al constructor de TwoDShape.
        style = ob.style;
    }

    double area() { // [*] Implemenatción del método abstracto area().
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("El triángulo es " + style);
    }
}

// Una subclase de TwoDShape para rectángulos.
class Rectangle extends TwoDShape {
    //Constructor predetermiando.
    Rectangle() {
        super();
    }

    // Constructor de Rectangle.
    Rectangle(double w, double h) {
        super(w, h, "Rectángulo"); // Invocar constructor de la superclase.
    }

    // Crear un cuadrado.
    Rectangle(double x) {
```

```
        super(x, "Cuadrado");
    }

    // Crear un objeto a partir de otro.
    Rectangle(Rectangle ob) {
        super(ob); // Pasar objeto al constructor de TwoDShape.
    }

    boolean isSquare() {
        if(getWidth() == getHeight()) return true;
        else return false;
    }

    double area() { // Sobrecarga de método.
        return getWidth() * getHeight();
    }
}

class Abstract {
    public static void main(String args[]) {
        TwoDShape shapes[] = new TwoDShape[4]; // Polimorfismo.

        shapes[0] = new Triangle("Delineado", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle(7.0);

        for(int i = 0; i < shapes.length; i++) {
            System.out.println("El objeto es " + shapes[i].getName());
            System.out.println("El área es " + shapes[i].area());

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

El ejemplo anterior devuelve:

```
El objeto es triangle
El área es 48.0

El objeto es Cuadrado
El área es 100.0

El objeto es Rectángulo
El área es 40.0

El objeto es triangle
El área es 24.5
```

Como muestra el ejemplo, todas las subclases de TwoDShape deben reemplazar area(). Si no se hace se produce un error en tiempo de compilación, de hecho el propio IDE avisa de ello.

Además, es totalmente válido que una clase abstracta contenga métodos concretos (no abstractos: showDim() y getName()) que una subclase pueda usar como tales. Solo los métodos declarados como abstract deben ser reemplazados en las subclases.

Palabra reservada 'final'.

A pesar de la potencia y utilidad del reemplazo de me´todos y de la herencia, en ocasiones habrá que evitarlos. Por ejemplo, cuando nos e quiera que el usuario de una clase pueda reemplazar el método de inicialización (piense en un dispositivo hardware, por ejemplo). Sea cual sea el motivo, en Java puede impedir que un método se reemplace o que una clase se herede por medio de la palabra clave 'final'.

Para evitar que un método se reemplace, especifique 'final' como modificador al inicio de su declaración. Los métodos declarados como final no se pueden reemplazar. Veamos un ejemplo

[Final.java]

```
// No se puede reemplazar un método final.

class A {
    final void meth() {
       System.out.println("Este es un método final");
    }
}

class B extends A {
    void meth() { // Cannot override the final method from A
        System.out.println("Ilegal"); // Error: No se puede reemplazar.
    }
}
```

No se puede heredar una clase 'final'.

```
final class C {
    // ...
}

// La siguiente clase no es válida.
class D extends C { // The type D cannot subclass the final class C
    // ...
}
```

Puede evitar que una clase se herede si precede su declaración con 'final'. Al declarar una clase como final se declaran todos sus métodos, implícitamente como final. No se puede declarar una clase como abstract y final, dado que una clase abstract está incompleta por sí misma y depende de sus subclases para proporcionar la implementación completa.

Utilizar final con miembros de datos.

Además de los usos anteriores de final, también se puede aplicar a variables miembro para crear algo parecido a constantes con nombre. Si añade final por delante del nombre de una variable de clase, su valor no se puede cambiar mientras dure el programa.

La clase Object.

Java define la clase especial Object, una superclase implícita de todas las demás clases. Es decir, todas las clases son subclases de Object, lo que significa que una variable de referencia de tipo Object puede hacer referencia a un objeto de cualquier clase. Además, como las matrices se implementan como clases, una variable de tipo Object también puede hacer referencia cualquier matriz.

Object define una serie de métodos que veremos más adelante, lógicamente todos ellos están disponibles para todos los objetos. Por ejemplo los métodos:

Object clone()	Crea un nuevo objeto idéntico al clonado.
String toString()	Devuelve cadena que describe al objeto.
boolean equals(Object objeto)	Determina si un objeto es igual a otro.
...	

PAQUETES E INTERFACES

Paquetes.

Los paquetes son grupos de clases relacionadas. Permiten organizar el código y constituyen otro nivel de encapsulación.

Los paquetes, por un lado, ofrecen un mecanismo para organizar como una unidad las piezas relacionadas de un programa. Debe accederse a las clases definidas en un paquete a través de su nombre de paquete. Por tanto un paquete ofrece una forma de denominar a una colección de clases. Por otra parte un paquete participa en el mecanismo de control de acceso de Java. Las clases definidas en un paquete pueden convertirse en privadas para ese paquete y ser inaccesibles para el código externo al mismo. Por ello un paquete constituye un medio para encapsular clases.

Los paquetes permiten dividir el espacio de nombres. al definir una clase en un paquete, se añade el nombre de dicho paquete a cada clase, lo que evita colisiones de nombres con otras clases del mismo nombre, pero en diferentes paquetes.

Como un paquete suele incluir clases relacionadas, Java define derechos de acceso especiales. En un paquete, puede definir código que sea accesible para otro código del mismo paquete, peor no para código externo. Esto le permite crear grupos independientes de clases relacionadas con un funcionamiento privado.

Definir un paquete.

Todas las clases de Java pertenecen a un paquete. Si no se especifica una instrucción 'package', se usa el paquete predeterminado (o global). El paquete predeterminado carece de nombre, lo que lo hace transparente.

En la mayoría de los casos tendrá que definir uno o varios paquetes para su código. Para crear un paquete, incluya un comando package al inicio del archivo Java. Las clases declaradas en ese archivo pertenecerán al paquete especificado. Por ejemplo:

```
package mypack;
```

Java usa el sistema de archivos para gestionar los paquetes y almacena cada uno en su propio directorio. Por ejemplo, los archivos .class de las clases que declare como parte de mypack deben almacenarse en el directorio mypack.

Puede crear una jerarquía de paquetes. Para ello, basta con separar el nombre de cada paquete del anterior con un punto. El formato general de una instrucción de paquete multinivel es el siguiente:

```
package paquete1.paquete2...paqueteN;
```

Sabiendo esto. Indicaremos como compilar y ejecutar programas haciendo uso de la rutas de paquetes (directorios):

Compilar:

```
javac mypack/Paquete.java
```

Ejecutar:

```
java mypack.Paquete
```

Paquetes y acceso a miembros.

Los paquetes también participan en el acceso a miembros. La visibilidad de un elemento se determina por su especificación de acceso (private, public, protected y default) y el paquete en el que se encuentra. Los distintos niveles de acceso (visibilidad) son los siguientes:

Misma clase:
Private(Sí), Default(Sí), Protected(Sí), Public(Sí).

Subclase mismo paquete:
Private(No), Default(Sí), Protected(Sí), Public(Sí).

No subclase mismo paquete:
Private(No), Default(Sí), Protected(Sí), Public(Sí).

Subclase paquete diferente:
Private(No), Default(No), Protected(Sí), Public(Sí).

No subclase paquete diferente:
Private(No), Default(No), Protected(No), Public(Sí).

Veamos un ejemplo de dos paquetes: packages.bookpack y packages.bookpacktext y el acceso desde packages.bookpacktext a packages.bookpack.*/

[packages.bookpack.Book.java]

```java
// Paquete Book para acceso público.

package packages.bookpack;

// Book y sus miembros deben ser 'public' para poder usarse en otros paquetes.
public class Book {
    private String title;
    private String author;
    private int pubDate;

    // El constructor debe ser también 'public'.
    public Book(String t, String a, int d) {
        title = t;
        author = a;
        pubDate = d;
    }

    // Sus métodos deben ser públicos.
    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(pubDate);
        System.out.println();
    }
}
```

[packages.bookpacktext.UseBook.java]

```java
// Esta clase pertenece al paquete packages.bookpacktext.

package packages.bookpacktext;
```

```java
// Usar la clase Book de packages.bookpack.
public class UseBook {
    public static void main(String args[]) {
        // Cualificar Book con el nombre de su paquete
        packages.bookpack.Book books[] = new packages.bookpack.Book[2];

        books[0] = new packages.bookpack.Book("Java 7", "Schildt", 2012);
        books[1] = new packages.bookpack.Book("Java 9", "Schildt", 2018);
        for(int i=0; i < books.length; i++) books[i].show();
    }
}
```

Para usar la clase Book desde otro paquete, debe usar la instrucción import (se verá próximamente) o cualificar su nombre para incluir la especificación completa del paquete. Así se ha hecho en la clase UseBook del paquete packages.bookpacktext, donde se ha cualificado Book de forma completa. Sin esta especificación no se encontraría Book al intentar compilar UseBook.

Miembros protegidos: 'protected'.

El uso de 'protected' suele ser confuso. El modificador protected crea un miembro accesible desde su paquete y para las subclases de otros paquetes. Por tanto, un miembro protected está disponible para todas las subclases, pero sigue protegido de accesos arbitrarios por parte de código externo a su paquete. Veamos un ejemplo:

[packages.bookpack.BookProtectedMembers.java]

```java
package packages.bookpack;

// Convertir en protected las variables de instancia de Book.
public class BookProtectedMembers {
    protected String title;
    protected String author;
    protected int pubDate;

    public BookProtectedMembers(String t, String a, int d) {
        title = t;
        author = a;
        pubDate = d;
    }

    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(pubDate);
        System.out.println();
    }
}
```

[packages.bookpacktext.ProtectedDemo.java]

```java
// Ejemplo de Protected.
package packages.bookpacktext;

class ExtBook extends packages.bookpack.BookProtectedMembers {
```

```
    private String publisher;

    public ExtBook(String t, String a, int d, String p) {
        super(t, a, d);
        publisher = p;
    }
    public void show() {
        super.show();
        System.out.print(publisher);
        System.out.println();
    }

    public String getPublisher() { return publisher;}
    public void setPublisher(String p) { publisher = p;}

    /* Lo siguiente es correcto ya que una subclase puede acceder
     a un miembro protected, en este caso, de BookProtectedMemebers */
     public String getTitle() { return title; }
     public void setTitle(String t) { title = t; }
     public String getAuthor() { return author; }
     public void setAuthor(String a) { author = a; }
     public int getPunDate() { return pubDate; }
     public void setPubDate(int d) { pubDate = d; }
}

public class ProtectedDemo {
    public static void main(String args[]) {
        ExtBook books[] = new ExtBook[2];

        books[0] = new ExtBook("Java 7", "Schildt", 2012, "Anaya");
        books[1] = new ExtBook("Java 9", "Schildt", 2018, "Anaya");

        for(int i=0; i < books.length; i++) books[i].show();

        // The field BookProtectedMembers.title is not visible.
        //books[0].title = "Test título"; // Error: No es accesible.
        }
}
```

En el ejemplo anterior, como ExtBook amplía Book, tiene acceso a los miembros protected de Book, aunque se encuentre en otro paquete. Por tanto, puede acceder directamente a title, author y pubDate, como hace en los métodos de acceso que crea para dichas variables. Sin embargo, en ProtectedDemo, se niega el acceso a estas variables ya que ProtectedDemo no es una subclase de Book. La última línea del código de esta clase daría error en tiempo de compilación.

Importar paquetes: 'import'.

Por medio de import puede revelar uno o varios miembros de un paquete, lo que permite usarlos directamente, sin la cualificación explícita. Su uso, mediante un ejemplo, es el siguiente:

```
import mypack.Myclass;
import mypack.*;
```

La biblioteca de clases estándar de Java.

La biblioteca de clases estándar de Java se incluye en paquetes. Esta biblioteca

de clases suele denominarse API (Interfaz de programación de aplicaciones) de Java. El API de Java se almacena en paquetes; en la parte superior de la jerarquía de paquetes se encuentra java. A partir de java encontramos otros subpaquetes:

java.lang	Clases de propósito general.
java.io	Clases de E/S.
java.net	Clases compatibles con redes.
java.applet	Clases para crear applets.
java.awt	Clases compatibles con AWT[*].

[*](Herramientas de ventana abstracta).

El paquete java.lang es exclusivo ya que se importa automáticamente en todos los programas de Java. Los demás paquetes es necesario importarlos explícitamente.

Interfaces.

En la programación orientada a objetos resulta muy útil definir lo que debe hacer una clase pero no cómo debe hacerlo. Un ejemplo de ello es el método abstracto, que define la firma de un método pero no proporciona su implementación. El método abstracto especifica la interfaz al método, pero no la implementación. Este concepto se puede desarrollar un paso más allá. En Java, puede separar la interfaz de una clase de su implementación por medio de la palabra clave reservada 'interface'.

Las interfaces son sintácticamente similares a las clases abstractas, pero en una interfaz los métodos carecen de cuerpo [*]. Tras definir una interfaz puede implementarse en todas las clases que se desee. Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) de los métodos descritos por la interfaz.

[*] En JDK 9 se puede añadir una implementación predeterminada a un método de interfaz, Además, admite métodos estáticos y private. Es decir, ahora, 'interface' puede especificar comportamientos.

En términos generales, una interfaz se puede definir como un contenedor que almacena las firmas de los métodos a implementar en el segmento de código. Una interfaz en Java es una colección de métodos abstractos y propiedades constantes. En las interfaces se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.

En una interfaz, los métodos son implícitamente public. Las variables no son variables de instancia, sino public, final y static de forma implícita, y deben inicializarse. Por tanto, son constantes.

Al declarar un interfaz como public, debe encontrarse en un archivo con el mismo nombre. Veamos un ejemplo de definición de interfaz:

[packages.interfaces.InterfaceSeries.java]

```java
/* Esta interfaz se declara como public para que se pueda implementar por el
código de cualquier paquete. */
public interface InterfaceSeries {
    int getNext();
    void reset();
    void setStart(int x);
}
```

Tras definir una interfaz, una o varias clases pueden implementarla. Para implementar una interfaz debe incluir la clausula 'implements' en una definición de clase y después crear los métodos definidos en la interfaz.

Los métodos que implementan la interfaz deben declararse como public. Además, la firma del método debe coincidir exactamente con la firma especificada en la definición de la interfaz.

Veamos un ejemplo. Se implementa la interfaz InterfaceSeries anterior. Se crea la clase ByTwos, que genera una serie de números:

[packages.interfaces.ByTwos.java]

```
package packages.interfaces;

class ByTwos implements InterfaceSeries  {

     int start;
     int val;

     ByTwos() {
          start = 0;
          val = 0;
     }

     // Implementación de los métodos de InterfaceSeries.

     public int getNext() {
          val += 2;
          return val;
     }

     public void reset() {
          val = start;
     }

     public void setStart(int x) {
          start = x;
          val = x;
     }
}
```

Veamos una clase que ilustra ByTwos:

[packages.interfaces.SeriesDemo.java]

```
package packages.interfaces;

public class SeriesDemo {
    public static void main(String args[]) {
        ByTwos ob = new ByTwos();

        for(int i=0; i < 5; i++)
            System.out.println("El siguiente valor es: " + ob.getNext());

        System.out.println("\nReiniciando");
        ob.reset();
        for(int i=0; i < 5; i++)
```

```
            System.out.println("El siguiente valor es: " + ob.getNext());

        System.out.println("\n Iniciando en 100");
        ob.setStart(100);
        for(int i=0; i < 5; i++)
            System.out.println("El siguiente valor es: " + ob.getNext());
    }
}
```

Métodos predeterminados de Interfaz.

Un método predeterminado de interfaz se define de forma similar al de un método por parte de una clase. La principal diferencia es que la declaración se precede de la palabra clave reservada 'default'.

[packages.interfaces.DefaultMethods.java]

```
package packages.interfaces;

public interface DefaultMethods {
    /* Una declaración de método de inetrfaz normal,
     no define una implementación predetermianda.*/
     int getUserID();

     /* Un método predeterminado. Proporciona una implementación
     predetermianda.*/
     default int getAdminID() {
        return 1;
     }
}
```

Como getAdminID() incluye su propia implementación predeterminada, no es necesario que la clase de implementación la reemplace. Es decir, si una clase de implementación no proporciona su propia implementación, se utiliza la predeterminada.

[packages.interfaces.MyIFImp.java]

```
package packages.interfaces;

class MyIFImp implements DefaultMethods {
    /* Solo hay que implementar getUserID() definido por DefaultMethods.
    Se puede permitir que getAdminID() sea predeterminado.*/
    public int getUserID() {
        return 100;
    }

}
```

Ahora mostramos un código que crea una instancia de MyIFImp y la utiliza para invocar tanto a getUserID() como getAdminID().

[packages.interfaces.DefaultMethodsDemo.java]

```
package packages.interfaces;
```

```
// Usa el meodo predetermiando de DefaultMethods
class DefaultMethodsDemo {
    public static void main(String args[]) {
        MyIFImp ob = new MyIFImp();

        /* Puede invocar getUserID() ya que se implementa explícitamente
         * por MyIFImp
         */
        System.out.println("Usuario ID es " + ob.getUserID());

        /* También puede invocar getAdminID(), debido a la implementación
         * predetermianda.
         */
        System.out.println("Administrador ID es " + ob.getAdminID());
    }
}
```

Es posible y habitual que una clase de implementación defina su propia implementación de un método predeterminado. Por ejemplo, aquí se reemplaza getAdminID(), se muestra a continuación.

```
class MyIFImp2 implements DefaultMethods {
    // Implementación para getUserID() y getAdminID().
    public int getUserID() {
        return 100;
    }

    public int getAdminID() {
        return 42;
    }
}
```

Al invocar a getAdminID() devuelve un valor distinto al predeterminado.

IMPORTANTE: Existe una diferencia fundamental entre una clase y una interfaz. Una clase puede mantener información de estado (a través de variables de instancia) pero una interfaz no.

Usar métodos estáticos en una interfaz.

Como sucede con una clase, un método estático definido por una interfaz se puede invocar de forma independiente a cualquier objeto. Por lo tanto no se necesita una implementación de la interfaz ni una instancia de la misma para invocar el método estático. La forma de invocar un método estático de una interfaz es similar a la forma de invocar un método estático de una clase. Veamos un ejemplo:

```
public interface StaticMethodInterface {
    /* Una declaración de método de inetrfaz normal,
    no define una implementación predetermianda.*/
    int getUserID();

    /* Un método predeterminado. Proporciona una implementación
    predetermianda.*/
    default int getAdminID() {
       return 1;
    }
    /* Un método de interfaz static. */
```

```
    static int getUniversalID() {
       return 0;
    }
}
```

La forma de invocar al método getUnievrsalID() es la siguiente:

```
      int uID = StaticMethodInterface.getUniversalID();
```

CONTROLAR EXCEPCIONES

Fundamentos del control de excepciones.

Una excepción es un error producido en tiempo de ejecución. En Java, todas las excepciones se presentan por medio de clases. Todas las clases de excepción derivan de Throwable. Por tanto, cuando se produce un error una excepción, se genera un objeto de un tipo de clase de excepción. Throwable tiene dos subclases directas: 'Exception' y 'Error'.

Las excepciones de tipo Error están relacionadas con errores producidos por la propia máquina virtual e Java y no en el programa. No se pueden controlar. Los errores de la actividad de un programa se representan por medios de subclases de Exception. Por ejemplo, la división por cero, los límites de una matriz y los errores de archivo. Por lo general, su programa debe controlar las excepciones de este tipo. Una subclase importante de Exception es RuntimeException, que se usa para representar distintos tipos de errores en tiempo de ejecución.

El control de excepciones en Java se controla a través de cinco palabras clave: try (intento), catch (atrapar), throw (tirar), throws y finally (finalizar).

Los instrucciones se incluyen en el bloque try. Si se produce una excepción en este bloque try, se genera un excepción. Su código puede capturarla a través de catch y con ello procesarla. Para generar una excepción manualmente se debe usar la palabra throw. El código que deba ejecutarse tras salir de un bloque try se incluye en un bloque finally.

try y catch forman la base del control de excepciones. Estas palabras clave funcionan de forma conjunta; no se puede usar catch sin try.

```
try {
    // Bloque de código para monitorizar errores.
}
catch (TiposExcepción1 obExcep) {
    // Controlardor para TipoExcepción1
}
catch (TiposExcepción2 obExcep) {
    // Controlardor para TipoExcepción2
}
...
```

Si el tipo de excepción especificado por una instrucción catch coincide con el de esa excepción, se ejecuta la instrucción catch (el resto se ignora). Al capturar una excepción obExcep recibe su valor. Si no se genera una excepción, el bloque try termina con normalidad y se ignoran todas las instrucciones catch.

[ExceptionDemo.java]

```
public class ExceptionDemo {
    public static void main(String args[]) {
        int nums[] = new int[4];

        try {
            System.out.println("Antes de que la excepción se genere.");

            // Generar una excepción de índice que supera los límites.
            nums[7] = 10;
            System.out.println("Esto no se mostrará.");
        }
        // Capturar errores de límite de matriz.
```

```
        catch (ArrayIndexOutOfBoundsException exec) {

            // Capturar la excepción.
            System.out.println("Índice fuera de rango.");
        }

        System.out.println("Después de la senatncia catch.");
    }
}
```

El código que permite monitorizar errores se incluye dentro de un bloque try. La excepción se genera fuera del bloque try y se captura por catch. Tras ello el control pasa a catch (La instrucción print() nunca se ejecuta) y se termina el bloque try. Es decir, no se invoca a catch sino que recibe la ejecución del programa.

En el programa anterior es importante 'ArrayIndexOutOfBoundsException', el tipo especificado para el tipo de excepción, que siempre debe coincidir, en caso contrario la excepción no se captura y se provocaría una finalización anómala del programa.

Utilizar varias instrucciones catch.

[ExceptionCatch.java]

```
// varias instrucciones catch.

public class ExceptionCatch {
    public static void main(String args[]) {
        // Aquí number ea mayor que denom.
        int number[] = {4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = {2, 0, 4, 4, 0, 8};

        for (int i=0; i < number.length; i++) {
            try {
                System.out.println(number[i] + "/" + denom[i] + " es " +
                                   number[i]/denom[i]);
            }
            // Varias instrucciones catch.
            catch (ArithmeticException exec) {
            // Capturar la excepción.
            System.out.println("No se puede dividir por cero.");
            }
            catch (ArrayIndexOutOfBoundsException exec) {
                // Capturar la excepción.
                System.out.println("Elemento no encontardo.");
            }
        }
    }
}
```

La salida del programa es la siguiente:

```
4/2 es 2
No se puede dividir por cero.
16/4 es 4
32/4 es 8
```

```
No se puede dividir por cero.
128/8 es 16
Elemento no encontrado.
Elemento no encontrado.
```

Cada instrucción catch solo responde a su propio tipo de excepción. Por lo general, las expresiones catch se comprueban en el orden que aparezcan en el programa. Solo se ejecuta la instrucción que coincida y los demás bloques catch se ignoran.

Anidar bloques try.

Un bloque try se puede anidar dentro de otro. Una excepción generada en el bloque try interno que no se capture con la instrucción catch asociada a dicho bloque se propaga al bloque try externo. Por ejemplo en este caso no se captura en la clausula catch interna, pero sí en la externa.

[ExceptionNestTrys.java]

```
// Un bloque try anidado.

public class ExceptionNestTrys {
    public static void main(String args[]) {
        // Aquí number ea mayor que denom.
        int number[] = {4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = {2, 0, 4, 4, 0, 8};

        try { // try externo.
            for (int i=0; i < number.length; i++) {
                try { //try anidado
                    System.out.println(number[i] + "/" + denom[i] + " es " +
                                       number[i]/denom[i]);
                }
                catch (ArithmeticException exec) {
                    // Capturar la excepción.
                    System.out.println("No se puede dividir por cero.");
                }
            }
        }
        catch (ArrayIndexOutOfBoundsException exec) {
            // Capturar la excepción.
            System.out.println("Elemento no encontardo.");
            System.out.println("Error Fatal - Programa terminado.");
        }
    }
}
```

En este ejemplo, una excepción que se puede controlar en el bloque try interno, en este caso un error de división por cero, permite al programa continuar. Sin embargo, el bloque try externo captura un error de límite de matriz, lo que hace que el programa termine.

MÓDULOS

Fundamentos de los módulos.

Los módulos ofrecen una forma de describir las relaciones y dependencias del código que comprende una aplicación. También permiten decidir qué partes del código son accesibles para otros módulos y cuáles no. Mediante el uso de módulos podemos crear programas escalables más fiables. La API de Java está organizada en módulos, y aunque estos están pensados para grandes proyectos, los programas pequeños también pueden beneficiarse de su uso.

En su sentido más básico, un módulo es una agrupación de paquetes y recursos a los que se puede hacer referencia conjuntamente a través del nombre del módulo.

Una declaración de módulo está contenida en un archivo llamado module-info.java. Por tanto, un módulo se define en un archivo fuente de Java. Luego javac compila ese archivo en un archivo de clase y se conoce como descriptor de módulo. El fichero module-info.java debe contener solo una definición de módulo, no es un archivo de uso general.

Una declaración de módulo empieza con la palabra clave 'module' y tiene esta forma general:

```
module nombreMódulo {
      // definición del módulo
}
```

Aunque una declaración de módulo puede estar vacía (lo que da como resultado una declaración que simplemente nombra el módulo), por lo general especifica una o varias clases que definen las características del módulo.

Palabras reservadas 'requires' y 'exports'.

Un módulo puede especifica que depende de otro. El uso de la instrucción requires especifica una relación de dependencia. Por defecto, la presencia de un módulo necesario se comprueba en tiempo de compilación y en el de ejecución. Un módulo también tiene capacidad para controlar cuál de sus paquetes (si es el caso) son accesibles para otros módulos. Esto se consigue con la palabra clave exports, Los tipos public y protected dentro de un paquete son accesibles para otros módulos solo si se exportan explícitamente.

Existe una convención para denominar a los módulos y es usar el método del nombre de dominio invertido, que consiste en usar al revés el nombre del dominio que "posee" el proyecto como prefijo para el módulo. Por ejemplo, un proyecto asociado a s2a.es podría usar es.s2a como prefijo del módulo. Lo mismo sirve par a los nombres de paquetes.

La siguiente aplicación define dos módulos: appstart y appfuncs.

appstart contiene un paquete llamado appstart.mymodappdemo que define el punto de entrada de la aplicación en una clase denominada MyModAppDemo, por consiguiente contiene el método main() de la aplicación.

appfuncs contiene un paquete llamado appfuncs.simplefuncs que incluye la clase SimpleMathFuncs. Esta clase define tres métodos static que implementan unas sencillas funciones matemáticas.

Toda la aplicación está contenida en un directorio que empieza mymodapp.

[SimpleMathFuncs.java]

```java
// Funciones matemáticas.

package appfuncs.simplefuncs;

public class SimpleMathFuncs {

    // Determina si a es factor de b.
    public static boolean isFactor(int a, int b) {
        if((b % a) == 0) return true;
        else return false;
    }

    // Devuelve el factor positivo más pequeño que a y b tienen en común.
    public static int lcf(int a, int b) {
        // Factor que utiliza valores positivos.
        a = Math.abs(a);
        b = Math.abs(b);

        int min = a < b ? a : b;

        for(int i = 2; i <= (min/2); i ++) {
            if(isFactor(i, a) && isFactor(i, b))
                return i;
        }

        return 1;
    }

    // Devuelve el factor positivo más grande que a y b tienen en común.
    public static int gcf(int a, int b) {
        // Factor que utiliza valores positivos.
        a = Math.abs(a);
        b = Math.abs(b);

        int min = a < b ? a : b;

        for(int i = (min/2); i >= 2 ; i --) {
            if(isFactor(i, a) && isFactor(i, b))
                return i;
        }

        return 1;
    }
}
```

[MyModAppDemo.java]

```java
// Ilustra una aplicación sencilla basada en módulos.

package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;

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

        if(SimpleMathFuncs.isFactor(2, 10))
            System.out.println("2 es factor de 10");
```

```
        System.out.println("El mínimo factor común de 35 y 105 es " +
                           SimpleMathFuncs.lcf(35, 105));

        System.out.println("El máximo factor común de 35 y 105 es " +
                           SimpleMathFuncs.gcf(35, 105));
    }
}
```

[module-info.java]

```
// Definición de módulo para el módulo de funciones.

module appfuncs { // Define un módulo para appfuncs.
    // Exporta el paquete appfuncs.aimplefuncs.
    exports appfuncs.simplefuncs;
}

// Definición de módulo para el módulo de la aplicación principal.

module appstart { // Define un módulo para appstart.
    // Exporta el paquete appfuncs.aimplefuncs.
    requires appmodules.appfuncs;
}
```

Para compilar esta aplicación seguir las instrucciones del libro Java 9 de Herbert Schildt (Java A Beginner's Guide, 7 ed.) en su capítulo módulos. Básicamente los pasos son los siguientes:

Compilación:

```
javac -d appmodules/appfuncs appsrc/appfuncs/module-info.java \
appsrc/appfuncs/appfuncs/simplefuncs/SimpleMathFuncs.java

javac --module-path appmodules -d appmodules/appstart \
appsrc/appstart/module-info.java
appsrc/appstart/appstart/mymodappdemo/MyModAppDemo.java
```

Ejecución:

```
java -module-path appmodules -m \
appstart/appstart.mymodappdemo.MyModAppDemo
```

java.base y los módulos de la plataforma.

Los módulos de la API se conocen como módulos de plataforma y todos los nombres empiezan con el prefijo java. Algunos ejemplos son: java.base, java.desktop y java.xml. Al modularizar la API, es posible implementar una aplicación con solamente los paquetes que requieren en vez de con todo el JRE. Teniendo en cuenta el tamaño del JRE, es una buena mejora. java.base, por ejemplo, exporta los paquetes esenciales de Java, como java.lang, java.io y java.util, entre otros. Dado su importancia es posible acceder a java.base desde cualquier módulo. No hace falta incluir una instrucción 'requires java.base' en una aplicación y declaración de módulo.