



UNIDAD 4. Programación Modular, Clases y Objetos.

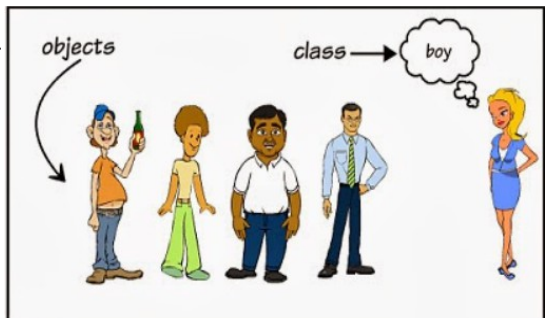
UNIDAD 4

PROGRAMACIÓN MODULAR, CLASES Y OBJETOS

1. PROGRAMACIÓN MODULAR EN JAVA.
2. MÉTODOS.
 - 2.1 Definir Métodos.
 - 2.2 Llamar a Métodos.
 - 2.3 Variables Globales y Locales.
 - 2.4 Parámetros.
 - 2.5 Sobrecarga de Métodos.
 - 2.6 Parámetros Objeto Mutables e Inmutables.
 - 2.7 Pasar Parámetros al Programa.
 - 2.8 Devolver valores.
3. PRECONDICIONES, POSTCONDICIONES Y ASERCIONES.
 - 4.1. Disparar Excepciones (throw).
 - 4.2. Aserciones (Assert).
4. RECURSIVIDAD.
5. API'S Y PACKAGES.
6. CLASES Y OBJETOS.
 - 6.1. Diferencias entre Objeto y Clase.
 - 6.2. Getters y Setters.
 - 6.3. Objetos Arrays y Strings.
7. CONSTRUCTORES E INICIALIZACIÓN.
 - 7.1. Inicializar Variables.
 - 7.2. Constructores.
 - 7.3. Recolección de Basura.
8. IMPLEMENTAR CORRECTAMENTE CLASES EN JAVA.
 - 8.1 La clase Object.
 - 8.2 Más Clases de Fábrica.
 - 8.3 Clases Inmutables y Singleton.
9. MÓDULOS.
10. MÁS ENUMERACIONES.
11. EJERCICIOS.

BIBLIOGRAFÍA:

- Java,cómo Programar (10ªEd) Pearson Paul Deitel y Harvey Deitel(2016).
- Intro. to Prog Using Java (7ª Ed). David J.Eck (2014)





UNIDAD 4. Programación Modular, Clases y Objetos.

4.1. PROGRAMACIÓN MODULAR EN JAVA.

La experiencia ha demostrado que la mejor manera de desarrollar y mantener un programa extenso es construirlo a partir de pequeñas piezas sencillas o **módulos**. Esta técnica se llama **divide y vencerás**.

En muchos lenguajes de tercera generación los procedimientos y funciones son la principal herramienta modular que se usa para crear esos trozos de código que agrupan sentencias. Además, suelen ofrecer alguna forma de agrupar estos trozos de código en módulos o librerías cuando están relacionados entre sí.

En los lenguajes orientados a objetos, a las subrutinas (procedimientos y funciones) se les llama métodos y las clases los agrupan. Java no permite definir métodos fuera de una clase.

Para escribir programas en Java, se combinan los nuevos métodos y clases que define el programador, con los métodos y clases predefinidos en la **Interfaz de Programación de Aplicaciones de Java** (la **API de Java** o **biblioteca de clases de Java**) y en otras librerías de clases realizadas por otros desarrolladores. Por lo general, **las clases relacionadas están agrupadas en paquetes**, de manera que se pueden *importar* a los programas y reutilizarse. **Los paquetes se organizan en módulos.**

La API de Java proporciona una gran colección de clases predefinidas que contienen métodos para realizar cálculos matemáticos, manipulaciones de cadenas y caracteres, operaciones de entrada/salida, acceso a bases de datos, operaciones de red, procesamiento de archivos, comprobación de errores y más...

Nota: cuanto más familiarizado estés con la API de Java (consultarla: <http://docs.oracle.com/javase/12/docs/api/>) más



UNIDAD 4. Programación Modular, Clases y Objetos.

fácilmente sacarás adelante tus programas, porque mucho del trabajo que tengas que realizar ya lo tienes programado => No reinventar la rueda implica reducir el tiempo de desarrollo y evita que se introduzcan errores de programación.

Las clases y sus métodos nos ayudan a dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas. Las instrucciones en los cuerpos de los métodos se escriben sólo una vez, se ocultan de otros métodos y se reutilizan en muchos programas.

Una razón para dividir un programa en módulos usando los métodos es *el enfoque divide y vencerás* que acabamos de mencionar. *Otra razón es la reutilización de código* (usar los métodos existentes como bloques de construcción para crear nuevos programas). A menudo se pueden crear programas a partir de métodos estandarizados, en vez de tener que crear código personalizado. Por ejemplo, en los programas anteriores no tuvimos que definir cómo leer datos del teclado, Java proporciona estas herramientas en la clase **Scanner**. Una tercera razón es *evitar la repetición de código*. El proceso de dividir un programa en métodos significativos hace que el programa sea más fácil de depurar y mantener.

Nota: *Para conseguir la reutilización de software, cada método debe limitarse a realizar una sola tarea bien definida y su nombre debe expresar esa tarea con efectividad. El motivo es que un método pequeño que lleva a cabo una tarea es más fácil de probar y depurar que uno más grande que realiza muchas tareas. Pero no es útil descomponer sin motivo una misma tarea.*

Nota: *si no puede elegir un nombre conciso que exprese la tarea de un método, tal vez esté tratando de realizar varias tareas en un mismo método o lo ha descompuesto en exceso.*



UNIDAD 4. Programación Modular, Clases y Objetos.

Un método se invoca mediante una llamada y cuando el método que se llamó completa su ejecución, devuelve el control y posiblemente un resultado, al método que lo llamó. Una analogía a esta estructura de programa es la forma jerárquica de la administración (figura 1). Un jefe (el solicitante) pide a un trabajador (el método llamado) que realice una tarea y que le devuelva los resultados después de completar la tarea. El método jefe no sabe cómo el método trabajador realiza sus tareas. Tal vez el trabajador llame a otros métodos trabajadores, sin que lo sepa el jefe. Este "ocultamiento" de los detalles de implementación fomenta la buena ingeniería de software.

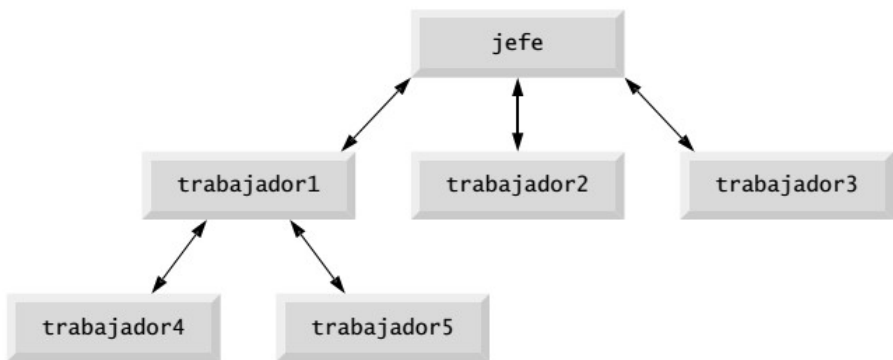


Figura 1: Cadena de llamadas (analogía con métodos de un programa)

Cuando se llama a un método que devuelve un valor que indica si el método realizó correctamente su tarea, debe comprobarse este valor de retorno y si avisa que no tuvo éxito, lidiar con el problema de manera apropiada.

SUBROUTINAS O MÉTODOS

Permiten romper la complejidad de un programa en trozos más manejables. Un método es un grupo de sentencias agrupadas, a las que se da un nombre y que al ejecutarse realizan una tarea. Su nombre se



UNIDAD 4. Programación Modular, Clases y Objetos.

usa cada vez que queramos realizar la tarea en un programa (llamada).

Los métodos pueden llamarse una y otra vez desde cualquier parte de un programa (y un método puede llamar a otros métodos). En Java los métodos pueden ser estáticos (contenidos en una clase) o no estáticos (los contienen los objetos creados a partir de la clase).

PRINCIPIO DE CAJA NEGRA

Se dice que algo es una caja negra porque no sabes lo que hay dentro de ella (o mejor dicho, no quieres saberlo), no te interesa ahondar en cómo realiza el trabajo que hace un método, solo quieres aprovecharte de que lo hace.

Pero una caja negra que no es capaz de interactuar con el mundo, no es útil. Por eso los métodos tienen una interfaz, un mecanismo que les permite recibir datos del mundo exterior y también devolver datos a quien los llama. Esta interfaz debe seguir unas reglas:

- **Bien definida y fácil de comprender:** ejemplos del mundo real pueden ser los electrodomésticos: puedes manejar una TV (encenderla, cambiar de canal..) sin necesidad de hacer un curso de postgrado en electrónica.
- **Permanente:** si cambia el interior pero no la interfaz, el cambio es transparente al exterior.

La interfaz de un método tiene una semántica y una sintaxis que hay que conocer. Para usarlo, necesitas comprender el trabajo que realiza (sin saber como lo hace), qué datos hay que pasarle, qué resultados te va a devolver y como se escribe (sintaxis) esa petición.

El contrato de una subrutina: Esto es lo que tienes que hacer para usarme y este es el trabajo que yo te haré...

Cuando no tengas métodos predefinidos (de fábrica) que hagan lo que



UNIDAD 4. Programación Modular, Clases y Objetos.

necesites, los lenguajes modulares te dan las herramientas para que implementes tus propios métodos.

4.2. MÉTODOS EN JAVA.

En POO a las subrutinas de clases y objetos se les llama **métodos**. En Java un método debe definirse dentro de una clase (muchos lenguajes de programación permiten también definir subrutinas independientes). En Java por tanto, subrutina y método son equivalentes. Además, una de las utilidades de las clases es **agrupar datos y métodos relacionadas**. Las clases a su vez, se agrupan en **paquetes**. Y los paquetes en módulos.

En una clase de Java pueden definirse **métodos estáticos** (pertenecen a la clase) y **no estáticos** (pertenecen a cada objeto que se cree a partir de esa clase). De hecho, cualquier cosa que se define a nivel de clase puede ser estática o no (las variables globales también).

4.2.1 DEFINICIÓN DE MÉTODOS.

La definición de un método tiene dos partes: la **declaración** y el **cuerpo**. La declaración consiste en definir la interface (es obligatorio indicar qué devuelve, su nombre y los paréntesis donde van los parámetros, aunque no tenga). Además de la declaración está el **cuerpo**, que es la implementación (las sentencias) que se ejecutan cada vez que se llame. **Sintaxis:** [algo] significa que algo es opcional, no es el operador array.

```
[modificadores] tipo_devuelto nombre ( [lista_parámetros] ) {  
    [sentencias;]  
}
```

Por ejemplo, el método **main()** de los programas que ya has creado



UNIDAD 4. Programación Modular, Clases y Objetos.

hasta ahora servirá como muestra. El bloque de sentencias que se ejecuta es el cuerpo del método, la implementación o el detalle de la caja negra.

MODIFICADORES

Son una o varias palabras reservadas del lenguaje Java que definen características de los elementos que modifican. No suelen ser obligatorios. Los que verás con mucha frecuencia son **static** que indica que el elemento es estático (de la clase) y **public** (controla el acceso). Aunque en realidad hay casi media docena y no solo se aplican a métodos, también a variables, clases y otros elementos.

Los **modificadores de acceso** son modificadores que controlan desde qué partes de un programa se puede utilizar el elemento (leerlo o cambiarlo si es un dato o llamarlo si es un método). Su significado:

- **public**: puede utilizarse desde cualquier parte de un programa.
- **protected**: accesible desde la propia clase donde se define, desde sus clases hijas y desde cualquier clase del mismo paquete donde está la clase en que se define.
- **Sin indicar/package**: al no indicar un modificador de acceso, usa el acceso por defecto (**package**) y el elemento solo puede ser accedido desde su clase y desde cualquier clase almacenada en el mismo paquete de la clase que lo contiene.
- **private**: se accede solamente desde la clase donde se define.

La palabra reservada **final** aplicada a la definición de una variable de tipo primitivo la convierte en constante. Solo se asigna valor una vez:

```
final double GRAVEDAD = 9.8;
```

Aplicada a la definición de una variable de tipo referencia hace que no se pueda cambiar de objeto, pero si se pueda hacer cambios al



UNIDAD 4. Programación Modular, Clases y Objetos.

contenido del objeto, si este es mutable.

```
class AC {  
    public static void main(String[] args) {  
        final StringBuilder c = new StringBuilder("Corriente");  
        System.out.println(c);  
        c.append(" alterna"); //cambiar estado interno del objeto  
        System.out.println(cadena);  
    }  
}
```

Aplicada a la definición de un método hace que no pueda ser modificado por una clase hija que lo herede. Bloqueas la sobreescritura:

```
class A {  
    final void m1() { System.out.println("Un método final."); }  
}  
  
class B extends A {  
    // ERROR: INTENTO DE SUSTITUIR(SOBREESCRIBIR) UN MÉTODO FINAL  
    void m1() {  
        System.out.println("Ilegal!!");  
    }  
}
```

Aplicada a un parámetro, hace que no puedas cambiar su valor dentro del código (aunque el parámetro sea una copia del parámetro actual, no puedes cambiar el valor). Ejemplo:

```
double cubo(final double n) {  
    n = n * n * n; // Error: n no puede cambiar }  
}
```

Por último, si aplicas final a una clase impides que se pueda extender, es decir, no puede tener clases hija (se hace estéril). Equivale a bloquear el mecanismo de la herencia.

```
public final class A { ... }  
public class B extends A { ... }: // Error: A no puede extenderse
```




UNIDAD 4. Programación Modular, Clases y Objetos.

TIPO DE DATO DEVUELTO

Cuando se define un método hay que pensar si necesitamos que tras hacer su trabajo debe equivaler a un dato (nos devuelve información y por tanto sea una función) o no (sea un procedimiento).

Si el método es una función (devuelve un valor) entonces **el valor que devuelve debe ser del tipo de dato indicado en su declaración**, como **String**, **int** o un array como **double[]** o un objeto de alguna clase.

Si no devuelve ningún valor (es un procedimiento) el tipo de dato que se indica en la declaración será **void** (vacío).

NOMBRE Y LISTA DE PARÁMETROS

Por último, **el nombre y la lista de parámetros son lo que define la interfaz del método**. Representan la información que se le entrega para que pueda hacer su trabajo. Por ejemplo, imagina que le dices a una TV que cambias de canal ¿Le dices de alguna forma a qué canal cambiar? Bueno, pues esa información es un parámetro. Un método puede tener cero o más parámetros separados por comas. Cada uno tiene un tipo de dato y un nombre por el que se le conoce dentro del código del método que aparece en su bloque. Ejemplo:

```
public void cambiaCanal( int numeroCanal ) { ... }
```

Cuando se quiera utilizar en un programa, la llamada será:

```
cambiaCanal(5);
```

Cuando tengas varios parámetros, aunque sean del mismo tipo, debes indicar el tipo de cada uno, es decir, no vale escribir:

```
metodo(double x, y) // Incorrecto!! Debes indicar tb. el tipo de y
```

Hay que indicar el tipo para cada uno:



UNIDAD 4. Programación Modular, Clases y Objetos.

(double x, double y) // Correcto!!

EJEMPLO 1: Varias definiciones de métodos en Java.

```
public static void jugarPartida() {
    // "public" y "static" son modificadores
    // "void" es lo que devuelve (nada -> es un procedure)
    // jugarPartida es el nombre del método
    // La lista de parámetros está vacía
    // Las declaraciones de datos propios y sentencias van aquí...
}

int dameSiguienteN(int n) {
    // No hay modificadores -> acceso package y método de objeto
    // devuelve un "int" -> es una función
    // "dameSiguienteN" es el nombre
    // Parámetros: tiene un parámetro de tipo int y nombre n
    // Las sentencias van aquí ...
}

static boolean menorQue(double x, double y) {
    // Modificadores: "static" -> es de la clase y acceso package
    // Devuelve boolean -> es una función
    // Parámetros: dos de tipo double que se llaman x e y
    // Las sentencias van aquí...
}
```

EJERCICIO 1: Completa la descripción de esta subrutina main():

```
public static void main(String[] args) { ... }
// Modificadores: _____
// Devuelve: _____
// Parámetros: _____
// Nombre de la subrutina: _____
```

4.2.2 LLAMANDO A LOS MÉTODOS.

Al definir un método, indicas que existe y lo que hace, pero **no se ejecuta hasta que se realice una llamada** (esto incluso es cierto para main(), que es llamado por la máquina virtual). Si tenemos este código



UNIDAD 4. Programación Modular, Clases y Objetos.

en el archivo Poker.java:

```
public class Poker {  
    public static void jugarPartida() {...}  
    // Otras cosas...  
}
```

Y este en el archivo Parchis.java:

```
public class Parchis {  
    public static void jugarPartida() {...}  
    // Otras cosas...  
}
```

La llamada a **jugarPartida()** puede hacerse desde la misma clase donde se define, por ejemplo desde su método **main()** o en otro de sus métodos. En este caso la llamada se escribe usando el nombre del método y pasándole los parámetros:

```
jugarPartida();
```

Pero en caso de llamarla desde otra clase diferente de donde se define, debes indicar a qué clase pertenece el método estático. Si queremos jugar una partida de póker la llamada será así:

```
Poker.jugarPartida();
```

Poner el nombre de la clase delante, ayuda a saber donde está el método, porque podrías tener otro método similar en la clase **Parchis**.

Así que para llamar a métodos estáticos normalmente se usan estas dos sintaxis:

```
nombre_método(parámetros);           // Desde su misma clase  
nombre_clase.nombre_método(parámetros); // Desde otra clase
```

Nota: Los métodos no estáticos, siempre necesitan el objeto que debe ejecutarlos, no la clase.



UNIDAD 4. Programación Modular, Clases y Objetos.

La ejecución de una llamada a un método acaba cuando:

- Ya no haya más sentencias que ejecutar.
- Se ejecuta una sentencia return [expresión];
- Se lanza una excepción que hace fallar el código.

EJEMPLO 2: Vamos a hacer un programa completo que incluya otro método además de main(). Vamos a programar un juego, de forma que el programa escoge al azar un número entre 1 y 100 y el usuario intenta adivinarlo. Si no lo consigue, el programa le indica si es menor o mayor. El método juegaPartida() controla el desarrollo de una partida de este juego. Y el método main() controla si se juega o no una partida.

```
package juegos;

import java.util.Scanner;

public class AdivinaNumero {
    static Scanner teclado;

    public static void main(String[] args) {
        teclado = new Scanner( System.in );
        System.out.println("Vamos a Jugar a que adivine un nº");
        System.out.println("entre 1 y 100.");
        boolean otraVez;
        do {
            juegaPartida(); // Llamada al método
            System.out.print("Juega otra vez? (S/N)");
            otraVez = teclado.next().equals("S")? true:false;
        } while (otraVez);
        System.out.println("Gracias por Jugar. Xao.");
    } // fin main()

    static void juegaPartida() {
        int numero; // El nº aleatorio
        int respuesta; // el nº indicado por el usuario
        int intentos; // nº de intentos

        numero = (int)(100 * Math.random()) + 1; // entre 1 y 100
        intentos = 0;
        System.out.println();
    }
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
System.out.print("Cuál es su primer valor? ");
while (true) {
    respuesta = teclado.nextInt(); // leer respuesta
    intentos++;
    if (respuesta == numero) {
        System.out.println("Acertado en " + intentos +
            " intentos! Ha ganado. El número era " + numero);
        break; // juego acabado
    }
    if (intentos == 6) {
        System.out.println("Ha perdido, límite 6 intentos.");
        System.out.println("El número era " + numero );
        break;
    }
    // Dar pistas
    if (respuesta < numero)
        System.out.print("Demasiado bajo. Pruebe otra vez: ");
    else if (respuesta > numero)
        System.out.print("Demasiado alto. Pruebe otra vez: ");
    }
    System.out.println();
} // fin juegaPartida()

} // fin clase AdivinaNumero
```

4.2.3 VARIABLES LOCALES Y GLOBALES.

Una clase además de definir métodos (el comportamiento) también puede declarar variables (el estado). Puedes declarar variables dentro de los métodos (son **variables locales** igual que los parámetros) y puedes hacerlo dentro de una clase pero fuera de cualquiera de sus métodos (**variables miembro o globales**). Si son estáticas (**variables de clase**) y en otro caso son de cada objeto creado a partir de la clase (**variables de instancia**).

VARIABLES GLOBALES

Las variables miembro de una clase pueden ser estáticas (de la clase) o no estáticas (de cada objeto de la clase). Una variable estática existe mientras lo haga la clase donde se define. Una clase comienza a existir



UNIDAD 4. Programación Modular, Clases y Objetos.

en el momento en que se usa por primera vez. Cualquier valor que se almacene en la variable estática es guardado en la clase. **Se puede utilizar desde cualquier método (estático o no) de la clase. Es un dato compartido y accesible desde todos los métodos de la clase.**

Una variable local, por contra, solo existe mientras se ejecutan las sentencias del método. Cuando la ejecución del método acaba, las variables locales desaparecen y son inaccesibles desde fuera (no existen).

La declaración de las variables miembro es igual que la de una variable local excepto por dos cosas:

- Se hace fuera de un método (aunque dentro de la clase).
- Puede tener modificadores de acceso (**public**, **protected**, **private**, **package**) y **static**.

Ejemplos:

```
static String nombreUsuario;  
public static int numeroJugadores;  
protected double velocidad, tiempo;
```

Una variable miembro estática que no es privada puede ser accedida desde fuera de la clase donde se declara además de desde dentro. Cuando se usa desde alguna otra clase hay que indicar el nombre de la clase si es estática:

```
clase.variable
```

Vamos a añadir dos variables miembro estáticas a la clase AdivinaNumero:

```
static int partidasJugadas;  
static int partidasGanadas;
```

Cuando declaras una variable local en un método, debes darle un valor inicial antes de usarla por primera vez. A las variables miembro sin



UNIDAD 4. Programación Modular, Clases y Objetos.

embargo, se les asigna un valor por defecto al declararlas. Los valores numéricos se inicializan a cero, las variables booleanas a false, las variables char con el carácter Unicode número cero y los objetos (como String) tienen de valor inicial por defecto null.

```
package juegos;

import java.util.Scanner;

public class AdivinaNumero {
    static Scanner teclado;
    static int partidasJugadas; // nº de partidas jugadas
    static int partidasGanadas; // nº de partidas ganadas

    public static void main(String[] args) {
        teclado = new Scanner( System.in );
        partidasJugadas = 0;
        partidasGanadas = 0;
        System.out.println("Vamos a Jugar a que adivine un nº");
        System.out.println("entre 1 y 100.");
        boolean otraVez;
        do {
            juegaPartida(); // Llamada a subrutina
            System.out.print("Juega otra vez? (S/N)");
            otraVez = teclado.next().equals("S")? true:false;
        } while (otraVez);
        System.out.println("Has jugado " + partidasJugadas + " veces, y has ganado " + partidasGanadas);
        System.out.println("Gracias por Jugar. Xao.");
    } // fin main()

    static void juegaPartida() {
        int numero; // El nº aleatorio
        int respuesta; // el nº indicado por el usuario
        int intentos; // nº de intentos

        numero = (int)(100 * Math.random()) + 1; // entre 1 y 100
        intentos = 0;
        System.out.println();
        System.out.print("Cuál es su primer valor? ");
        while (true) {
            respuesta = teclado.nextInt(); // leer respuesta
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
intentos++;
if (respuesta == numero) {
    System.out.println("Acertado en " + intentos +
        " intentos! Ha ganado. El número era " + numero);
    partidasGanadas++;
    break; // juego acabado
}
if (intentos == 6) {
    System.out.println("Ha perdido, límite 6 intentos.");
    System.out.println("El número era " + numero );
    break;
}
// Dar pistas
if (respuesta < numero)
    System.out.print("Demasiado bajo. Pruebe otra vez: ");
else if (respuesta > numero)
    System.out.print("Demasiado alto. Pruebe otra vez: ");
}
System.out.println();
partidasJugadas++;
} // fin juegaPartida()
} // fin clase AdivinaNumero
```

Las clases que no declaran un paquete están en el paquete por defecto, así que cualquier clase que no declare un paquete tendrá acceso a `partidasJugadas()`, `partidasGanadas()` y `juegaPartida()`.

Nota: Es buena costumbre hacer que las variables miembro sean privadas, salvo que haya un buen motivo para no hacerlo.

VARIABLES LOCALES

Dentro de un método puedes usar 3 tipos de variables: las variables locales (se declaran en el método), parámetros formales (son como variables locales) y las variables definidas a nivel de clase (declaradas fuera del método).

Las variables locales no son visibles ni existen fuera del método donde se declaran. Los parámetros se usan para enviar valores al método



UNIDAD 4. Programación Modular, Clases y Objetos.

cuando se llama, una vez que comienza su ejecución, son como variables locales. Los cambios que hagan las sentencias del método a los parámetros, no tienen efecto en el exterior (salvo que el parámetro sea un objeto no inmutable).

Pero si la variable se define fuera del método, los cambios que se realicen si permanecen cuando el método acaba. Cuando la variable existe independientemente del método, decimos que es global.

Nota: *Es mejor minimizar la cantidad de variables globales. Solo usamos datos globales cuando sea realmente necesario (se usan en varios métodos, describe el estado de los objetos, ...). Es preferible pasar los valores que necesite un método como parámetros que declararlos globales por ahorrar la definición.*

Un buen motivo para usar variables globales es tiene sentido que esos datos existan más allá de la ejecución de los métodos porque describen características de un objeto o son usados por varios métodos de la clase. Es más cómodo que esos métodos accedan a las variables de la clase que pasarlas como parámetros.

Cuando se ejecuta la declaración de la variable, se reserva memoria para almacenar su valor. Después de la declaración se realiza una inicialización (una asignación de un primer valor). Ejemplo:

```
int cuenta; // Declara una variable llamada cuenta
cuenta = 0; // se da un valor inicial
```

También es posible hacer las dos cosas en la misma sentencia:

```
int contador = 0; // Declara e inicializa
```

El valor puede ser cualquier expresión compatible con el tipo de dato de la variable declarada, no tiene porqué ser una constante. Además, se pueden declarar e inicializar varias variables en la misma sentencia:



UNIDAD 4. Programación Modular, Clases y Objetos.

```
char inicial1 = 'D', inicial2 = 'E';  
int x, y = 1;           // OK, pero solo se inicializa y!  
int N = 3, M = N + 2;  // OK, N se inicializa antes de usarla en M
```

En bucles es algo que se utiliza mucho, declarar e inicializar variables que se usan dentro del bucle:

```
for( int i = 0; i < 10; i++ ) { // La variable i se define fuera  
    System.out.println(i);  
}
```

Recuerda que eso es una abreviación equivalente a esta:

```
{  
    int i;  
    for( i = 0; i < 10; i++ ) {  
        System.out.println(i);  
    }  
}
```

INICIALIZAR VARIABLES GLOBALES

Una variable miembro también puede inicializarse en el momento de declarar la con un valor distinto al que tendría por defecto:

```
public class Banco {  
    private static double ratioInteres = 0.05;  
    private static double maxDescubierto = 200.0;  
    // resto de variables y métodos...  
}
```

Una variable estática se crea tan pronto como el Intérprete de Java carga la clase y la inicialización se produce en ese momento. Esta asignación es la única que puede ocurrir fuera de un método. Por tanto estas sentencias no son legales:

```
public class Banco {  
    private static double ratioInteres;  
    ratioInteres = 0.05; // ILEGAL!! SENTENCIA FUERA DE MÉTODO!!
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Si a una variable miembro no se le asigna un valor inicial, se utiliza un valor por defecto. Por ejemplo al declarar un entero sin inicializar, se le asigna el valor 0:

```
public class C1 {  
    static int contador;           // Inicialización a 0  
    static int contador1 = 0;     // Inicialización equivalente
```

Para inicializar un array al declararlo, debes aportar los valores separados por comas y encerrados entre llaves:

```
// Declarar y rellenar un array de 9 enteros  
int[] primerosPrimos = { 2, 3, 5, 7, 11, 13, 17, 23, 29 };
```

La otra forma de inicializar un array, se puede usar tanto al declararlo como después (en una sentencia de asignación), pero los elementos tendrán su valor por defecto (a null en el ejemplo):

```
String[] nombres = new String[100];
```

También es posible usar **bloques de código de inicialización** (fuera de cualquier método pero dentro de una clase) para inicializar tanto variables miembro de la clase (estáticas) o de los objetos.

Sin embargo para el caso de las variables de instancia (de los objetos) esta tarea se suele hacer en unos métodos dedicados especialmente a esta tarea llamados **constructores**. Ejemplo de código de inicialización:

```
public class E1 {  
    private static int deLaClase;  
    private int delObjeto;  
    static { deLaClase = 5; } // Se ejecuta 1 vez al cargar clase  
    { delObjeto = 3; }       // Se ejecuta cada vez que se cree objeto  
    // Más cosas ...  
}
```

4.2.4 PARÁMETROS O ARGUMENTOS.

Si un método es una caja negra, un parámetro es el mecanismo para



UNIDAD 4. Programación Modular, Clases y Objetos.

que el mundo exterior (quien realiza la llamada) indique la información con la que debe trabajar el método.

USANDO PARÁMETROS.

Como ejemplo, vamos a realizar el método que implementa el algoritmo secuenciaPedrisco(n) que genera e imprime la secuencia $3N+1$ (si n es impar, el siguiente valor se obtiene multiplicando el actual por 3 y sumando 1), y si es par, el siguiente valor se obtiene dividiéndolo por 2. Se continúa mientras el valor no sea 1. Por ejemplo, comenzando por $N=3$, la secuencia será: 3, 10, 5, 16, 8, 4, 2, 1. El método recibirá el número inicio donde comienza la secuencia:

```
/**
 * Imprime la secuencia 3N+1 por la salida estándar.
 * ENTRADAS: el valor inicial de la secuencia (entero positivo)
 */
static void secuenciaPedrisco(int inicio) {
    int N;          // Un término de la secuencia
    int contador;   // El número de términos
    N = inicio;     // El primer término a usar.
    contador = 1;   // El primer término
    System.out.println("La secuencia 3N+1 comienza en " + N);
    System.out.println();
    System.out.println(N); // el primer término se imprime
    while (N > 1) {
        if (N % 2 == 1) // si N es impar
            N = 3 * N + 1;
        else
            N = N / 2;
        contador++; // un nuevo término
        System.out.println(N); // imprime el término
    }
    System.out.println();
    System.out.println("Hay " + contador + " términos.");
} // fin secuenciapedrisco
```

El método tiene un parámetro llamado inicio de tipo entero. Dentro del método el parámetro se puede utilizar igual que cualquier otra variable



UNIDAD 4. Programación Modular, Clases y Objetos.

local. Pero el valor que contiene viene del exterior, se indica cuando se realiza la llamada. Por ejemplo, la clase donde esté el método puede contener un método main() que lo llame, pasándole un valor que indique un usuario al programa:

```
package series;

import java.util.Scanner;

public class Pedrisco {

    static Scanner teclado = new Scanner( System.in );

    public static void main(String[] args) {
        System.out.println("Usar la secuencia Pedrisco (3N+1)");
        System.out.println("comenzando en el valor que indiques.");
        System.out.println();
        int K; // Indicado por el usuario
        do {
            System.out.println("Valor inicial (0 para acabar): ");
            K = teclado.nextInt();      // Leer valor
            if (K > 0)
                secuenciaPedrisco(K);
        } while (K > 0); // Continúa solo si el nº K > 0
    } // fin main

    /**
     * Imprime la secuencia 3N+1 por la salida estándar.
     * ENTRADAS: el valor inicial de la secuencia (entero positivo)
     */
    static void secuenciaPedrisco(int inicio) {
        // Sentencias del código anterior...
    }
} // fin de la clase Pedrisco
```

PARÁMETROS FORMALES Y ACTUALES.

La palabra parámetro o argumento se utiliza para definir dos conceptos diferentes, pero relacionados. Hay parámetros que se usan cuando se define el método, como el parámetro inicio al definir



UNIDAD 4. Programación Modular, Clases y Objetos.

secuenciaPedrisco. Y hay parámetros que se usan en las llamadas a las subrutinas, como k en el código anterior. Los parámetros en la definición del método son los **parámetros formales**. Los parámetros en las llamadas son los **parámetros actuales**.

Un parámetro formal es un identificador y es similar a una variable, así que tiene un tipo de asociado. Un parámetro actual puede ser una expresión del tipo indicado en el parámetro formal o un tipo compatible (por ejemplo si el parámetro formal es double, el actual puede ser un int):

```
static void ejemplo(int N, double x, boolean prueba) {  
    // sentencias...  
}
```

Una llamada:

```
ejemplo(17, Math.sqrt(z+1), z >= 10);
```

Cuando se ejecute la llamada al método, el efecto sería como tener un bloque con las sentencias del método, salvo por las cuestiones del ámbito de las variables:

```
{  
int N; // Alojar memoria para los parámetros formales  
double x;  
boolean prueba;  
    N = 17; // rellenar valores con parámetros actuales  
    x = Math.sqrt(z+1);  
    test = (z >= 10);  
    // Sentencias...  
}
```

Nota: **Errores comunes con los parámetros de métodos** que cometen los programadores novatos y que deberían hacer preguntarse si ese dato es realmente un parámetro o una variable local del método o global de la clase:



UNIDAD 4. Programación Modular, Clases y Objetos.

- *Dar valor a un parámetro al principio del método -> pierdes el valor recibido del exterior ¿Realmente es un parámetro?*
- *Pedir al usuario el valor de un parámetro dentro de un método -> pierdes el valor recibido en la llamada ¿Realmente es un parámetro?*
- *No usar el parámetro en el código del método -> ¿Realmente es un parámetro?*
- *Usar en todas las llamadas el mismo valor -> ¿Realmente es un parámetro?*

4.2.5 SOBRECARGA DE MÉTODOS.

Para llamar a un método debes conocer su nombre, cuántos parámetros formales tiene y el tipo de dato de cada uno. Esta información se denomina la **firma del método**. Ejemplo:

```
ejemplo(int, double, boolean)
```

Observa que la firma no incluye los nombres de los parámetros formales ni el tipo de dato que devuelve.

Java permite que dos métodos diferentes definidos en la misma clase tengan el mismo nombre siempre que tengan firmas diferentes. Cuando esto ocurre se dice que **el nombre del método está sobrecargado** porque tiene diferentes códigos asociados. El intérprete averigua qué código ejecutar por la cantidad y el tipo de los parámetros actuales que se utilizan en la llamada.

Ya has usado métodos sobrecargados cuando has sacado valores por la consola con **System.out.println()**:

```
println(int)          println(double)      println(char)
println(boolean)      println()
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Así que cuando haces una llamada como `System.out.println(17)`, se utiliza `println(int)`, mientras que al hacer la llamada `System.out.println('A')` se utiliza la firma `println(char)`.

Nota: el tipo devuelto no está incluido en la firma, no puedes tener dos métodos en la misma clase con la misma firma, aunque devuelvan tipos de datos diferentes.

EJEMPLO 3: Escribir un método que imprima por pantalla todos los divisores de un número n.

```
/**
 * Imprimir todos los divisores de n
 * ENTRADAS: n (entero positivo)
 * SALIDAS: ninguna.
 */
static void imprimeDivisores( int n ) {
    int d; // uno de los divisores de n
    System.out.println("Los divisores de " + n + " son:");
    for ( d = 1; d <= N; d++ ) {
        if ( n % d == 0 ) // d divide a n?
            System.out.println(d);
    }
}
```

EJEMPLO 4: Escribe un método estático y privado de la clase **Ejem2**, que se llame **rellena** y tenga dos parámetros, uno de tipo carácter y otro entero e imprima una línea de texto con tantas repeticiones del carácter como valor tenga el parámetro entero. Si el valor es menor o igual de 0, imprime un salto de línea.

```
/**
 * Escribe una línea de texto que contiene n copias de la letra c
 * ENTRADAS: c y n
 * SALIDAS: ninguna
 */
private static void rellena( char c, int n ) {
    int i; // variable contador
    for ( i = 1; i <= n; i++ ) {
```




UNIDAD 4. Programación Modular, Clases y Objetos.

```

        System.out.print( c );
    }
    System.out.println();
}

```

EJEMPLO 5: haz un método en la clase Ejem2 llamado rellena que acepte de parámetro un String. Para cada carácter del String debe imprimir una línea con 25 veces ese carácter (usa el método rellena de ejemplo 4).

```

/**
 * Para cada carácter del String s, escribe una línea con
 * 25 copias del carácter.
 */
private static void rellena( String s ) {
    int i;    // variable contador
    for ( i = 0; i < s.length(); i++ ) {
        rellena( s.charAt(i), 25 ); // La versión anterior
    }
}

```

4.2.6 PARÁMETROS OBJETO MUTABLES E INMUTABLES.

Es posible pasar como un único parámetro a un método un array completo con todos sus valores. Por ejemplo, vamos a hacer un método llamado muestraLista() a la que pasamos un array a de números enteros y nos los muestra con el formato: "[i₀, i₁, i_n]".

```

static void muestraLista( int[] a ) {
    System.out.print('[');
    for ( int i = 0; i < a.length; i++ ) {
        if ( i > 0 ) System.out.print(', '); // el primero sin coma
        System.out.print( a[i] );
    }
    System.out.println(']');
}

```

La llamada debe usar un parámetro actual de tipo array de enteros:

```

int[] n = new int[3];
n[0] = 42;
n[1] = 17;

```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
n[2] = 256;  
muestraListaInt( n );
```

Genera la salida: [42, 17, 256] .

Los **arrays son objetos mutables**, así que cuando se pasan como parámetros, aunque se pasa una copia de la referencia, si se le hacen cambios dentro del método a los elementos, los cambios si son visibles al exterior, es decir, en el interior: no puedes cambiar la referencia del exterior (es una copia), pero si el contenido (no es una copia porque la copia de la referencia apunta al mismo objeto, a la misma RAM).

EJEMPLO 6: Diferencia al pasar un string (inmutable) o un array (mutable) a un método que los modifica.

```
public static void metodo(String s, int[] a) {  
    a[0] = -1;  
    s = s.toLowerCase();  
    a = new int[2];  
    a[0] = a[1] = 3;  
}  
:  
// Una llamada al método  
int[] array = {1, 2, 3};  
String cadena = "HOLA CARACOLA";  
metodo(cadena, array);  
System.out.println( "Cadena: " + cadena + " Array: " +  
                    Arrays.toString(array) );  
:
```

LA SALIDA ES: "Cadena: HOLA CARACOLA Array: [-1, 2, 3]"

4.2.7 PASAR PARÁMETROS AL PROGRAMA.

El método main() de un programa tiene un parámetro de tipo **String[]**. Cuando se llama al programa, se le pueden pasar parámetros a este método que quedan guardados como String en este array de cadenas. Si se llama desde el SO, este es quien le pasa este array. El array

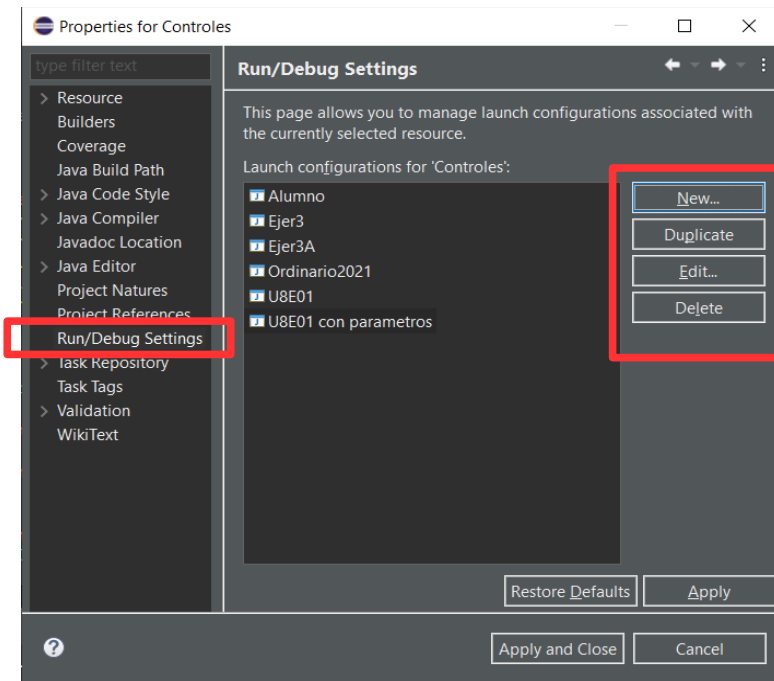


UNIDAD 4. Programación Modular, Clases y Objetos.

contiene los argumentos u opciones de la línea de comandos cuando se llama al programa. Por ejemplo si hacemos esta llamada desde línea de comandos:

```
c:\>java unPrograma -> No hay ítems en String[] llamado args  
c:\>java unPrograma uno dos tres -> args tiene 3 elementos
```

Si estamos en Eclipse, para hacer que el programa reciba parámetros debemos modificar la configuración de ejecución o crear una nueva si queremos tener varias formas de ejecutar el programa.

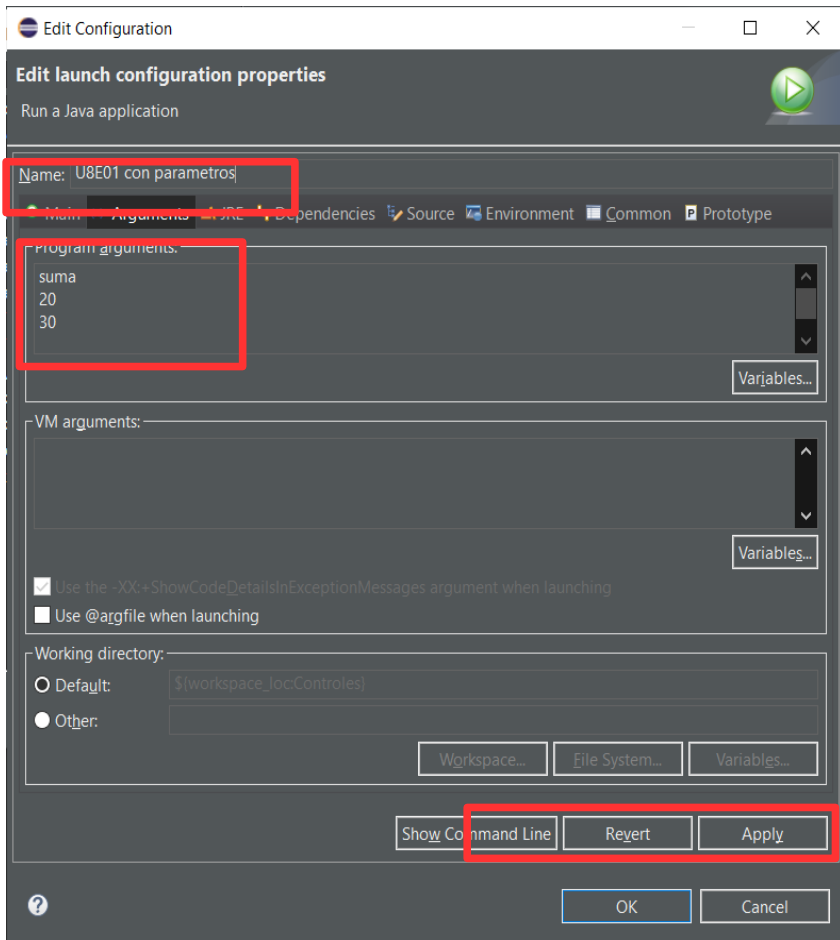


Para ello, debes ir al menú Proyecto -> Propiedades -> en la lista de la izquierda la opción Run/Debug Settings. En la ventana que aparece puedes seleccionar una de las configuraciones y modificarla, duplicarla, etc. o puedes crear una nueva con New.



UNIDAD 4. Programación Modular, Clases y Objetos.

En la siguiente figura se crea una nueva configuración llamada "U08E01 con parametros" y en la lista que tiene la etiqueta "Program arguments" se indican las cadenas que queremos pasarle al programa cuando se ejecute:



Luego al ejecutarlo, en el botón de ejecución puedes elegir una

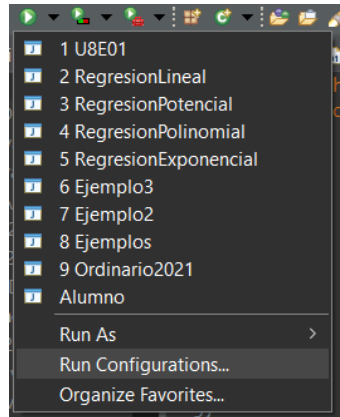


UNIDAD 4. Programación Modular, Clases y Objetos.

determinada configuración (los parámetros que recibe son solo una de las posibles configuraciones de cada ejecución).

O incluso en el mismo proyecto indicar la configuración que quieres ejecutar por defecto.

EJEMPLO 7: Programa que imprime los argumentos que se le pasan por línea de comandos.



```
public class CliDemo {  
  
    public static void main(String[] args) {  
        System.out.println("Usted ha tecleado " + args.length +  
            " argumentos");  
        if (args.length > 0) {  
            System.out.println("Son:");  
            for (int i = 0; i < args.length; i++)  
                System.out.println(" " + args[i]);  
        }  
    } // fin main()  
} // fin CliDemo
```

En la práctica los argumentos de la línea de comandos se usan frecuentemente para pasar nombres de ficheros u opciones de funcionamiento que queremos que el programa realice. Aunque en el caso de aplicaciones gráficas no se suele utilizar.

EJEMPLO 8: Si el programa no recibe un parámetro que necesita, sale indicando que le falta. (Otras opciones podrían ser pedirlo de forma interactiva).

```
public class CliDemo {  
    public static void main(String[] args) {  
        if (args.length != 1) {  
            System.out.println("Necesito un parámetro.");  
        }  
    }  
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
        System.exit(-1); // Sale con código de error -1
    }
    // Aquí el programa continuaría trabajando...
} // fin main()
} // fin CliDemo
```

4.2.8 NÚMERO VARIABLE DE PARÁMETROS.

En Java existe la posibilidad de definir métodos que tengan un número de parámetros variable (indefinido o desconocido). Esta posibilidad del lenguaje se llama VarArgs (**argumentos variables**) y aparece a partir del JDK SE 5.0.

Antes de J2SE 5.0 si necesitabas un método con una cantidad indeterminada de valores, se podía definir con un parámetro array (u otra colección) tal y como se hace en main(). Ejemplo:

```
int sumar(int[] nums) {
    int total = 0;
    for(int i = 0; i < nums.length; i++) total+= nums[i];
    return total;
}
```

para llamarlo, primero se crea un array con los valores que necesite y se lo pasa. Ejemplo:

```
int[] array = { 1, 2, 3, 4, 5 };
sumar(array);
```

Si no queremos usar arrays o colecciones podemos sobrecargar el método con varios números de parámetros. Pero es poco flexible y elegante, mejor lo del array:

```
int sumar( int a, int b) { return a + b; }
int sumar( int a, int b, int c) { return a + b + c; }
int sumar( int a, int b, int c, int d) { return a + b + c + d; }
...
```

La nueva sintaxis consiste en añadir al tipo del parámetro variable



UNIDAD 4. Programación Modular, Clases y Objetos.

tres puntos (...). Ejemplo:

```
int sumar(int... nums) {
    int total = 0;
    for(int i = 0; i < nums.length; i++) total+= nums[i];
    return total;
}
```

El tratamiento dentro del método es como si fuera un array. Pero en el programa que hace la llamada no se usan arrays:

```
sumar(1, 2);
sumar(1, 2, 3, 4, 5);
```

En la llamada nos ahorramos tener que crear el array, ya lo hace el compilador convirtiendo estas llamadas:

sumar(1, 2, 3, 4) en esta: **sumar(new int[] {1, 2, 3, 4})**

Argumentos variables mezclados con otros:

Cuando defines un método, solo puedes tener un parámetro vararg (con argumentos variables) y debe ir al final de la lista de parámetros.

EJEMPLO 9: Mostrar la tabla de multiplicar de un número n:

```
void tablaMultiplicar(int n, int... v) {
    System.out.println("Tabla de multiplicar de: " + n);
    for (int i=0 ; i < v.length; i++) {
        System.out.println(v[i] + " x " + n + " = " + (v[i] * n));
    }
}
...
// La llamada podría ser:
tablaMultiplicar(5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
tablaMultiplicar(3, 0, 1, 2, 3);
```

EJERCICIO 2: Crea un método que acepte dos o más valores enteros (como mínimo dos) e imprima el máximo valor.



UNIDAD 4. Programación Modular, Clases y Objetos.

4.2.8 DEVOLVER VALORES.

Un método que devuelve un valor es una función. Tienen la característica de que las llamadas a las funciones pueden aparecer en expresiones jugando el papel de cualquier valor. Por ejemplo pueden aparecer en las expresiones de las sentencias de control, en la parte derecha de las expresiones de asignación o en las expresiones que calculan el valor de un parámetro de la llamada a otro método.

Cuando en un método al acabar su ejecución quieres devolver un valor que sustituye a la llamada en el código, se usa la **sentencia return** seguida de una expresión. Sintaxis:

return [expresión];

Esta sentencia solamente puede aparecer dentro de un método y el tipo de dato de la expresión debe coincidir con el tipo de dato que se indica en la declaración del método. Ejemplo:

```
static double pitagoras(double c1, double c2) {  
    // Calcula la hipotenusa de un triángulo rectángulo de  
    // catetos c1 y c2  
    return Math.sqrt( c1 * c1 + c2 * c2 );  
}
```

Podemos usar llamadas como esta:

```
total = 17 + pitagoras(12, 5);
```

Cuando la sentencia **return** se ejecuta, el método acaba. Y aunque es habitual que aparezca al final de los métodos, realmente puede aparecer en cualquier parte y no solo una vez.

EJEMPLO 10: calcular el siguiente término de la secuencia pedrisco.

```
static int sigTermino( int actual ) {  
    if (actual % 2 == 1) return 3 * actual + 1;  
    else return actual / 2;
```




UNIDAD 4. Programación Modular, Clases y Objetos.

```
}
```

EJEMPLO 11: Una función equivalente con un solo return al final.

```
static int sigTermino(int actual) {  
    int respuesta;  
    if (actual % 2 == 1)  
        respuesta = 3 * actual + 1;  
    else  
        respuesta = actual / 2;  
    return respuesta;  
}
```

EJEMPLO 12: Otro equivalente a los anteriores.

```
static int sigTermino( int actual ) {  
    if (actual % 2 == 1) return 3 * actual + 1;  
    return actual / 2;  
}
```

Si un método se declara como procedimiento (devuelve void), también puede usar la sentencia **return**; (sin indicar expresión) para acabar su ejecución.

EJEMPLO 13: Un método que usa return para acabar su ejecución.

```
static void prueba(int n) {  
    if( n == 0 ) return; // Sale si el parámetro n es 0  
    System.out.printf("El inverso de %d es %.3f\n", n, 1.0 / n);  
}
```

EJERCICIO 3: Crea la clase Ejer3 que tenga un método estático que acepte dos o más valores enteros (como mínimo dos) y devuelva el máximo valor. Hazla ejecutable y prueba el método.

EJERCICIO 4: Modifica la clase Ejer3 y sobrecarga el método para que también pueda devolver la mayor cadena de texto (la máxima cadena de varias, como mínimo dos). Prueba el nuevo método también.



UNIDAD 4. Programación Modular, Clases y Objetos.

4.3. PRECONDICIONES Y POSTCONDICIONES.

Los programadores invierten una gran parte de su tiempo en mantener y depurar código. Para facilitar estas tareas y mejorar el diseño en general, se especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y postcondiciones respectivamente.

- **Precondiciones:** deben ser verdaderas cuando se llama a un método. Las precondiciones describen las restricciones en los parámetros de un método y cualquier otra expectativa que tenga el método en relación con el estado actual de un programa, justo antes de empezar a ejecutarse. Si no se cumplen las precondiciones, entonces el comportamiento del método es indefinido (puede lanzar una excepción, continuar con un valor ilegal o tratar de recuperarse del error). Nunca hay que esperar un comportamiento consistente si no se cumplen las precondiciones.
- **Postcondiciones:** Una postcondición es verdadera una vez que el método regresa con éxito. Las postcondiciones describen las restricciones en el valor de retorno, así como cualquier otro efecto secundario que pueda tener la ejecución del método.

Al definir un método, debes documentar todas sus pre y post condiciones.

Cuando no se cumplen sus precondiciones o postcondiciones, los métodos por lo general lanzan excepciones. También podrían informar al que hace la llamada de esta situación devolviéndoles un centinela o un valor incorrecto que debe comprobar quien llama. La diferencia es que con la excepción obligas a tratarla al que llama, mientras que avisando con un valor de retorno dejas a su buena



UNIDAD 4. Programación Modular, Clases y Objetos.

voluntad preocuparse del error (enfoque más propenso a cometer errores).

Nota: no tiene sentido que un método haga su trabajo y luego compruebe las precondiciones (si no se cumple la precondición, en el mejor de los casos el trabajo realizado es inútil).

```
// MAL ENFOQUE -> PELIGROSO O TRABAJO INÚTIL
```

```
método(parámetros)
```

```
    // realizo el trabajo...
```

```
    // Compruebo las precondiciones...
```

```
// ENFOQUE CORRECTO
```

```
método(parámetros)
```

```
    // Primero compruebo las precondiciones...
```

```
    // Una vez comprobadas, realizo el trabajo porque es útil...
```

Como ejemplo, piensa en el método estático **charAt()** del objeto **String**, que tiene un parámetro **int** que es un índice que indica la posición que ocupa en el **String** el carácter que quieres. Vamos a pensar en las precondiciones y postcondiciones que le vendrían bien. Como precondición, el índice debe ser mayor o igual a 0 y menor que la longitud del **String**. Si no se cumple se lanza la excepción **IndexOutOfBoundsException**. Postcondiciones no necesita, si se cumple la precondición, el resultado será correcto.

EJEMPLO 14: Precondición de **getLetra(String s, int posicion)**.

```
public static char getLetra(String s, int pos) throws Exception {  
    if( s == null || pos >= s.length() || pos < 0 )  
        throw new Exception("No existe posicion " + pos);  
    return s.charAt(pos);  
}
```

EJEMPLO 15: Precondición de **getLetra(String s, int posicion)** devolviendo el carácter **'\0'** si no se ha podido obtener. En este caso el que llama debe preocuparse de comprobar si el valor devuelto es



UNIDAD 4. Programación Modular, Clases y Objetos.

correcto ¿Se hará siempre que sea necesario? -> más peligroso.

```

public static char getLetra(String s, int pos) {
    if( s == null || pos >= s.length() || pos < 0) return (char)0;
    return s.charAt(pos);
}

```

4.3.1 LANZANDO (THROWING) EXCEPCIONES.

El contrato de un método consiste en que quien realice la llamada envíe valores apropiados en los parámetros, pero ¿qué ocurre si el que llama no cumple este compromiso? Lo normal es que el método lance una excepción. Por ejemplo: `Double.parseDouble("Hola");` lanza una excepción porque **no estoy pasando en el String un valor convertible a double, y se lanza `NumberFormatException`.**

Podrías querer hacer lo mismo en tus métodos. Java tiene la sentencia que te permite lanzar excepciones producidos por valores de parámetros incorrectos, la excepción es de tipo **`IllegalArgumentException`**:

```

throw new IllegalArgumentException( mensaje_de_error );

```

EJEMPLO 16: Añadir una precondición en el código del ejercicio `secuenciaPedrisco` de forma que cuando el número del parámetro inicio sea menor o igual a cero, lance una excepción.

```

static void secuenciaPedrisco(int inicio) {
    if (inicio <= 0) // contrato incumplido!
        throw new IllegalArgumentException( "Inicio debe ser >= 0");
    // sentencias...
}

```

A veces se confunden las palabras reservadas **throw** y **throws** así que volvemos a explicar como se utilizan. Imagina que hacemos un método en el que validamos ciertos datos y si alguno sobrepasa cierto valor podemos lanzar una excepción, ya sea creada por nosotros mismos o cualquiera del tipo `java.lang.throwable`, por ejemplo:



UNIDAD 4. Programación Modular, Clases y Objetos.

```
1 public void pruebaThrow() throws Exception {
2     Exception exception = new Exception();
3     int a = 98;
4     int b = 101;
5
6     if (a > 100 || b > 100) {
7         throw exception;
8     }
9 }
```

En la línea 7 lanzamos la excepción creada en la línea 2. Al hacerlo, si pruebas el programa observarás que la aplicación se detiene.

La palabra reservada **throws** sirve para indicar que un método lanza una excepción de un tipo específico o general, se puede utilizar si la excepción se va manejar con un try-catch o no. Así se utiliza:

```
1 public static void prueba() throws SQLException {
2     // Manejamos la excepcion de tipo SQL
3 }
4
5 public static void prueba() throws NullPointerException {
6     // Manejamos la excepcion de tipo NULL
7 }
8
9 public static void prueba() throws DataFormatException {
10     // Manejamos la excepcion de tipo DataFormat
11 }
```

En el primer ejemplo de `throw` ya usamos `throws` para especificar que ese método puede lanzar una excepción. De no hacerlo, el compilador nos obligaría a tratarla dentro del método, pero no es lo que queremos, queremos que le llegue la excepción a quien realice la llamada cuando no cumple las precondiciones, queremos que en su código se preocupe de cumplir esas precondiciones o capturar el error en otro caso.

4.3.2. ASERCIONES.

Condiciones que se deben cumplir en cualquier parte del código. Sirven para asegurar la validez de un programa al atrapar los errores



UNIDAD 4. Programación Modular, Clases y Objetos.

potenciales e identificar posibles errores lógicos **durante el desarrollo**. Las precondiciones y las postcondiciones son dos tipos de aserciones. Las precondiciones son aserciones sobre el estado de un programa a la hora de invocar un método y las postcondiciones son aserciones sobre el estado de un programa cuando el método termina.

Aunque las aserciones pueden establecerse como comentarios para guiar al programador durante el desarrollo del programa, Java incluye dos versiones de la instrucción **assert** para validar aserciones mediante programación. La instrucción **assert** evalúa una expresión booleana y, si es false, lanza una excepción **AssertionError** (una subclase de Error). La primera forma de la instrucción assert es:

```
assert expre; // lanza AssertionError si expre es false
```

La segunda forma es:

```
assert expre1:expre2; /* evalúa expre1 y lanza excepción  
AssertionError con expre2 como el mensaje de error, en caso  
de que expre1 sea false */
```

Puedes utilizar aserciones para implementar las precondiciones y postcondiciones mediante programación, o para verificar cualquier otro estado intermedio que te ayude a asegurar que el código esté funcionando en forma correcta.

EJEMPLO 17: Comprobar validez de código mediante aserciones

```
// Comprobar mediante assert que un valor esté dentro del rango.  
import java.util.Scanner;  
  
public class PruebaAssert {  
    public static void main(String[] args) {  
        Scanner teclado = new Scanner(System.in);  
        System.out.print("Escriba un numero entre 0 y 10: ");  
        int n = entrada.nextInt();
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
// asegura que el valor sea >= 0 y <= 10
assert (n >= 0 && n <= 10) : "numero incorrecto: " + n;
System.out.printf("Usted escribio %d\n", numero) ;
}
} // fin de la clase PruebaAssert
```

Nota: Los usuarios no deben encontrar ningún error tipo **AssertionError**. Se usan sólo durante el desarrollo del programa. De hecho, para poder verlas hay que ejecutar el programa con la opción **enable assertion** (-ea), ejemplo:

```
java -ea Programa.java
```

Nunca se debe atrapar una excepción tipo **AssertionError**. No debemos usar la instrucción **assert** para indicar problemas en tiempo de ejecución en el código de producción (el que se entrega al cliente), debemos usar el mecanismo de las excepciones para eso. Las aserciones solo durante el desarrollo.

EJERCICIO 5: Crea un método llamado **media** que acepte de parámetro un array de enteros y devuelva un **double** con el valor de la media.

- Implementa con aserciones la precondition de que el array no sea null y tenga al menos dos elementos.
- Si el código debemos entregarlo, cambia las aserciones por excepciones.

4.4 RECURSIVIDAD.

Es una técnica para resolver un problema que se basa en que el problema se puede expresar como una solución inicial y el resto de soluciones se basan en resolver el mismo problema modificado hasta llegar a esa solución inicial.



UNIDAD 4. Programación Modular, Clases y Objetos.

EJEMPLO 18: La potencia positiva de un número es base elevado a exponente donde exponente es un entero positivo (mayor o igual que cero). Si exponente es 0, el resultado es 1. En otro caso:

base * base * ... (multiplicar base exponente veces)

Así que $2^4 = 2 * 2 * 2 * 2$ (2 multiplicado 4 veces por él mismo).

En Java podemos resolverlo con un bucle:

```
/* precondition: exponente >= 0 */
potencia = 1.0;
for(int i= 0; i < exponente; i++) { potencia *= base; }
```

Pero si expresamos el problema de forma recursiva, vemos que el caso inicial es $base^0$ que siempre devuelve el valor 1. El resto de casos se pueden calcular de la misma manera hasta llegar al caso inicial. Eso expresado de forma recursiva:

$$\text{Potencia}(b,e) = \begin{cases} 1, & \text{si } e = 0 \text{ (caso inicial)} \\ b * \text{potencia}(b, e-1) & \text{en otro caso (general)} \end{cases}$$

Es decir: $\text{potencia}(2,4)$

$$\begin{aligned} &= 2 * \text{potencia}(2, 3) \\ &= 2 * 2 * \text{potencia}(2, 2) \\ &= 2 * 2 * 2 * \text{potencia}(2,1) \\ &= 2 * 2 * 2 * 2 * \text{potencia}(2,0) \\ &= 2 * 2 * 2 * 2 * 1 \\ &= 2 * 2 * 2 * 2 \end{aligned}$$

- **PROS:** mecanismo muy potente y elegante.
- **CONTRAS:** menos eficiente que versiones no recursivas del mismo algoritmo.

EJERCICIO 6: Piensa como expresar la multiplicación de un número entero y positivo $n1$ por otro número entero y positivo $n2$ usando solo



UNIDAD 4. Programación Modular, Clases y Objetos.

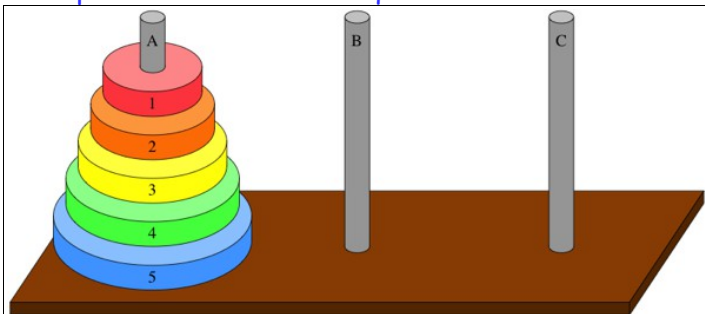
operaciones suma.

- Haz un método llamado producto que acepte los dos números a multiplicar (controla precondiciones) y lo implementas con un bucle.
- Piensa como expresar el problema de forma recursiva.
- Crea el método producto2 que resuelva el problema de forma recursiva.

EJEMPLO 19: Las Torres de Hanoi. Este problema se suele plantear a menudo en programación, especialmente para explicar la recursividad. En su forma más tradicional, consiste en 3 postes verticales. En uno de los postes se apila un número indeterminado de discos perforados por su centro (elaborados de madera), que determinará la complejidad de la solución. Por regla general, solucionar el problema manualmente con 7 discos ya es complicado.

Los discos se apilan sobre uno de los postes en tamaño decreciente de abajo a arriba. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio -desde la base del poste hacia arriba- en uno de los postes (A), quedando los otros dos postes vacíos (B y C). El juego consiste en pasar todos los discos desde el poste ocupado (es decir, el que posee la torre) a uno de los otros postes vacíos. Para realizar este objetivo, es necesario seguir tres simples reglas:

- Solo se puede mover un disco cada vez.
- Un disco de mayor tamaño no puede estar sobre uno más pequeño.
- Solo se puede mover el disco que se encuentre arriba.





UNIDAD 4. Programación Modular, Clases y Objetos.

Figura 2: Juego de las Torres de Hanoi.

Aunque el origen del juego es de un matemático francés, se suele adornar con leyendas como:

“En el gran templo de Benarés, debajo de la cúpula que marca el centro del mundo, yace una base de bronce, donde se encuentran acomodadas tres agujas de diamante, cada una del grueso del cuerpo de una abeja y de una altura de 50 cm. En una de estas agujas, Dios, en el momento de la creación, colocó sesenta y cuatro discos de oro -el mayor sobre la base de bronce, y el resto de menor tamaño conforme se va ascendiendo-. Día y noche, incesantemente, los sacerdotes del templo se turnan en el trabajo de mover los discos de una aguja a otra de acuerdo con las leyes impuestas e inmutables de Brahma, que requieren que siempre haya algún sacerdote trabajando, que no muevan más de un disco a la vez y que deben colocar cada disco en alguna de las agujas de modo que no cubra a un disco menor. Cuando los sesenta y cuatro discos hayan sido transferidos de la aguja en la que Dios los colocó en el momento de la creación, a otra aguja, el templo y los brahmanes se convertirán en polvo y junto con ellos, el mundo desaparecerá...”

Existen diversas formas de llegar a la solución final, todas ellas siguiendo estrategias diferentes.

UNA SOLUCIÓN RECURSIVA.

Si numeramos los discos desde 1 hasta n , si llamamos *origen* a la primera pila de discos (izquierda), *destino* a la tercera (derecha) y *auxiliar* a la intermedia (centro), y si a la función la denomináramos *hanoi()*, con *origen*, *auxiliar* y *destino* como parámetros, el algoritmo de la función sería el siguiente:

Algoritmo Torres de Hanói



UNIDAD 4. Programación Modular, Clases y Objetos.

Entrada: 3 pilas de n°s llamadas *origen*, *auxiliar*, *destino*

Salida: La pila *destino*

SI *origen == 1*

ENTONCES

mover el disco 1 de pila *origen* a la pila *destino*

SI NO

// mover todas las fichas menos la más grande (n) a auxiliar

hanoi(origen, destino, auxiliar)

// mover la ficha grande hasta la varilla final

mover disco *n* a *destino*

// mover las fichas restantes, 1...n-1, encima de la grande (n)

hanoi(auxiliar, origen, destino)

FIN SI

El número de movimientos mínimo a realizar para resolver el problema de este modo es de 2^{n-1} , siendo *n* el número de discos. El programa en Java:

```
package TorresHanoi;

import javax.swing.JOptionPane;

public class TorresDeHanoi {
    private int NumeroDeDiscos; // Nº de discos
    private int NumeroDeMovimientos; // Nº movimientos realizados

    public int getNumeroDeDiscos(){ return NumeroDeDiscos; }

    public void setNumeroDeDiscos(int NumeroDeDiscos) {
        this.NumeroDeDiscos = NumeroDeDiscos;
    }

    public int getNumeroDeMovimientos() {
        return NumeroDeMovimientos;
    }

    public void setNumeroDeMovimientos(int NumeroDeMovimientos) {
        this.NumeroDeMovimientos = NumeroDeMovimientos;
    }
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
}

public void Captura(){
    NumeroDeDiscos = Integer.parseInt(
        JOptionPane.showInputDialog("¿CUANTOS DISCOS?")
    );
}

public void Intercambio(int NumDiscos, char A, char B, char C){
    // Los parámetros se envían a la segunda clase
    /* Ten en cuenta que el parámetro A, toma el lugar de
    la torre inicio, la que inicialmente contiene todos los
    discos; el parámetro B, será la torre auxiliar; y el
    parámetro C será la torre destino, donde quedarán
    al final del juego todas las fichas */
    if (NumDiscos==1){
        /* si el número de discos es igual a uno, lógicamente se
        moverá el disco de la torre inicio directamente a la
        de destino*/
        setNumeroDeMovimientos( getNumeroDeMovimientos() + 1 );
        JOptionPane.showMessageDialog(null, "Mover disco " +
            NumDiscos + " de la torre " + A + " a la torre "
            + C + "\nMOVIMIENTOS: " + NumeroDeMovimientos);
    }
    else{
        /* se realizarán mas movimientos y serán los siguientes*/
        Intercambio(NumDiscos-1, A, C, B);
        /*... y entonces el método se llama a sí mismo, moviendo
        primero, el disco de la torre A (inicio) a la torre C
        (destino) */
        setNumeroDeMovimientos(getNumeroDeMovimientos()+1);
        JOptionPane.showMessageDialog(null, "Mover disco " +
            NumDiscos + " de la torre " + A + " a la torre " + C +
            "\nMOVIMIENTOS: " + NumeroDeMovimientos);
        /*... el método vuelve a llamarse a sí mismo, esta vez
        los parámetros donde se encontraba la variable
        A ahora estará la variable B, para indicar el siguiente
        movimiento*/
        Intercambio(NumDiscos-1, B, A, C);
    }
}

public void Movimientos() {
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
JOptionPane.showMessageDialog(null, "TOTAL DE MOVIMIENTOS: "
                                + NumeroDeMovimientos);
    }
}
```

Esta otra clase será la que tenga la subrutina main():

```
package TorresHanoi;

import javax.swing.JOptionPane;

public class Jugar {

    public static void main( String arg[] ) {
        TorresDeHanoi k;
        k= new TorresDeHanoi();
        // se declara y crea la variable de la primera clase
        k.Captura();
        k.Intercambio( k.getNumeroDeDiscos(), 'A', 'B', 'C');
        // se reciben los parámetros del método Intercambio
        k.Movimientos();
    }
}
```

SOLUCIÓN NO RECURSIVA

Otra manera de resolver el problema, sin utilizar la recursividad, se basa en el hecho de que para obtener la solución más corta, es necesario mover el disco más pequeño en todos los pasos impares, mientras que en los pasos pares solo existe un movimiento posible que no lo incluye. El problema se reduce a decidir en cada paso impar a cuál de las dos pilas posibles se desplazará el disco pequeño. El algoritmo en cuestión depende del número de discos del problema:

Nota: Eliminar la recursividad de un algoritmo que resuelve fácilmente (pero de forma no eficiente) un problema, suele ser una de los ejercicios típicos cuando estudias algoritmos.



UNIDAD 4. Programación Modular, Clases y Objetos.

4.5 API'S Y PACKAGES.

Puedes escribir programas con una sencilla interfaz de usuario que utilice la consola. Pero las aplicaciones con GUI tienen ventanas, botones, barras de desplazamiento, menús, cajas de edición de texto y otros muchos elementos que hacen más sencilla la utilización al usuario pero obligan al programador a manejar un abanico mucho más amplio de posibilidades. El programador experimenta un crecimiento importante de la complejidad al aparecer un mayor número de métodos para gestionar la interfaz de usuario y otros propósitos.

Toolboxes (Cajas de Herramientas)

Alguien que necesite programas para ordenadores Macintosh—generar programas que se vean y comporten de la forma que el usuario espera—tendrán que usar el Macintosh Toolbox, una colección de cientos de diferentes métodos que le permitirán abrir ventanas, dibujar figuras y texto, añadir botones a las ventanas, responder a eventos del ratón, etc. Hay otras rutinas para crear menús y reaccionar a las opciones que elija el usuario. Al margen de la GUI, necesitará rutinas para trabajar con ficheros, comunicarse por la red, enviar información a una impresora, comunicarse con otros programas y hacer muchas otras cosas que las aplicaciones necesitan hacer. Microsoft Windows ofrece su propio conjunto de métodos a los programadores y son un poco diferentes de las que usan los sistemas Mac. GNU/Linux también tiene diferentes Toolboxes para que los usen los programadores.

Cada proyecto necesita una mezcla de innovación y reutilización de herramientas existentes. Un programador que tenga un conjunto de herramientas para usar, comenzará con estas herramientas preconstruidas en el lenguaje como variables, sentencias de asignación, sentencias de control, etc.



UNIDAD 4. Programación Modular, Clases y Objetos.

A esto el programador añade toolboxes con cientos de métodos ya creados que realizan ciertas tareas. Si estas herramientas están bien diseñadas, pueden utilizarse como cajas negras: puedes solicitarles un trabajo aunque no sepas como lo hacen.

La parte de innovación consiste en coger todas estas herramientas y aplicarlas a un proyecto concreto que resuelva un problema (procesar textos, almacenar cuentas y operaciones bancarias, procesar imágenes, navegar por la web, juegos de ordenador, etc.). Esto es programar aplicaciones.

API

Una toolbox de software es un tipo de caja negra que ofrece una interfaz al programador. La interfaz es una especificación de los métodos que tiene, los parámetros que hay que pasarles y las tareas que realizan. **Esta información es la API (Application Programming Interface) asociada a la toolbox.**

El lenguaje Java ofrece una gran API estándar. Ya hemos utilizado partes de esa API como las subrutinas matemáticas (`Math.sqrt()` por ejemplo) y la salida de datos por consola (`System.out.print()`). El API estándar de Java incluye rutinas para trabajar con GUI, comunicaciones de red, lectura y escritura de ficheros, etc.

Como Java es independiente de la plataforma (hardware y sistema operativo) el mismo programa debe ejecutarse en Mac OS, Windows, Linux y otros entornos. La misma API debe funcionar en todas las plataformas. Pero lo que realmente es independiente de la plataforma es la interfaz, no la implementación. Si te aprendes y usas la API, tendrás programas que funcionen en muchas plataformas.

PACKAGE



UNIDAD 4. Programación Modular, Clases y Objetos.

Los métodos de la API de Java se agrupan en clases. Para ofrecer una organización a gran escala, las clases se pueden agrupar en paquetes. **Un paquete también puede contener a otros paquetes además de clases.** Toda la API estándar de Java está implementada en varios paquetes. Uno de ellos se llama **java** y contiene muchos paquetes que no son para GUI y el paquete original para GUI que se llama "AWT".

Otro package, "javax", se ha añadido a la versión 1.2 y contiene las clases usadas por Swing, una toolbox para GUI.

Un package puede contener clases y otros paquetes. El paquete contenido se llama un "sub-package." Tanto el paquete **java** como **javax** contienen subpaquetes. Uno de los subpaquetes de java es "awt". Su ruta completa es: **java.awt**. Aunque ya no se utiliza intensivamente, AWT contiene algunas clases como **Graphics** que contiene las rutinas para dibujar en la pantalla, **Color** para definir colores y **Font** para trabajar con texto de forma gráfica. Sus rutas completas serán: **java.awt.Graphics**, **java.awt.Color** y **java.awt.Font**.

De igual forma, **javax** contiene un subpaquete llamado **javax.swing**, que incluye clases para la GUI como **javax.swing.JButton**, **javax.swing.JMenu** y **javax.swing.JFrame**. Las clases de javax.swing, junto con las clases básicas de java.awt, son toda la parte de la API de Java que hace posible escribir programas GUI en Java.

El paquete **java** contiene otros subpaquetes como **java.io** (para entrada/salida de datos), **java.net** (comunicaciones de red) y **java.util** (una variedad de clases de utilidad). Otro subpaquete básico está en **java.lang** (contiene clases como String, Math, Integer y Double). Si miras la organización de forma gráfica, quizás sea más sencilla.

Por ejemplo, la documentación de la API de Java 8 tiene más de 210



UNIDAD 4. Programación Modular, Clases y Objetos.

paquetes con más de 4025 clases.

Muchas de ellas son oscuras o muy especializadas, nadie las domina todas, como mucho las investigas si alguna vez las necesitas. Si quieres ojearlas puedes usar este enlace:

<http://download.oracle.com/javase/8/docs/api/>

Si cambias el 8 de la URL por otra versión probablemente vayas a la documentación de la API de esa versión de java.

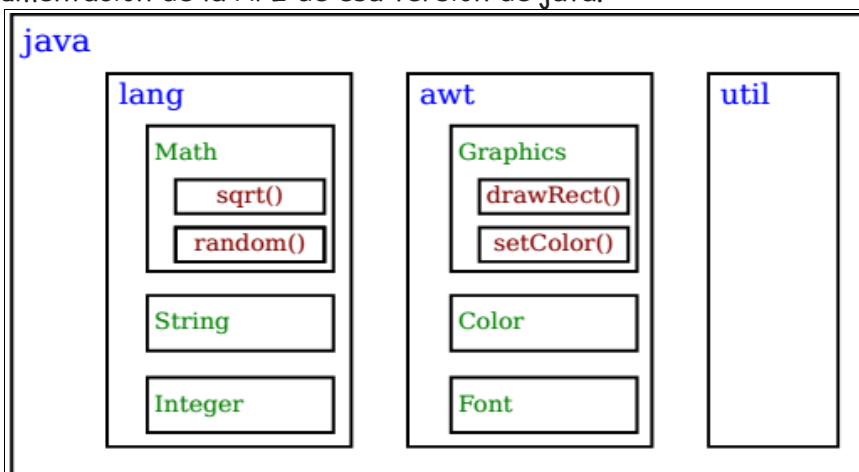


Figura 3: Esquema de algunos paquetes de la API estándar de Java.

USANDO CLASES DE LOS PAQUETES: `import`

Imagina que quieres usar la clase `java.awt.Color` en un programa. Como cualquier clase, `java.awt.Color` es como un tipo (puedes usarlo para declarar variables y parámetros). Una forma de hacerlo es usar la ruta completa a la clase. Por ejemplo si quieres declarar una variable `color1`:

```
java.awt.Color color1;
```

Para no tener que usar las rutas completas a las clases puedes importar las clases de un paquete al principio del fichero de código



UNIDAD 4. Programación Modular, Clases y Objetos.

fuente y posteriormente solamente debes poner el nombre de la clase, sin la ruta. Por ejemplo:

```
import java.awt.Color;
...
Color color1;
```

Nota: las sentencias **import** aparecen al principio del fichero de código fuente (como mucho detrás de la sentencia **package** si la hay) y fuera de cualquier clase.

Más que una sentencia, **import** es una directiva. Se puede usar una abreviatura para importar todas las clases bajo un paquete, que es el carácter asterisco y significa todas las clases del paquete.

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

Nota: no importa las clases de subpaquetes.

Nota: algunos programadores piensan que usar el asterisco es una mala opción porque importas clases que no usas, que es mejor importar solo aquellas clases que usas de forma individual. Que no te sepa mal usarlo, es lo habitual.

Por ejemplo es muy común ver programas GUI que hagan esto:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Si debes ser cuidadoso al usar el asterisco al importar en estos casos:

- Que dos clases se llamen igual en diferentes paquetes importados con asterisco. Por ejemplo: `java.awt` y `java.util` contienen una clase llamada **List**. Si importas `java.awt.*` y `java.util.*`, al usar solamente el nombre `List`, se produce una ambigüedad. En este caso, debes:



UNIDAD 4. Programación Modular, Clases y Objetos.

- Usar el nombre completo de la clase (java.awt.List o java.util.List).
- No importar con asterisco si solo necesitas una de las dos.

Como las clases del paquete **java.lang** son fundamentales, se importan automáticamente. Así que es como si cada programa Java comenzase con la sentencia:

```
import java.lang.*;
```

También podemos importar los métodos y miembros estáticos de una clase con **import static clase.*** y no tener que prefijar en el código la clase a la que pertenecen, aunque si el mismo elemento está en dos clases diferentes, debemos usar el prefijo de clase para eliminar la ambigüedad. Ejemplo:

```
import static java.lang.System.*;
import static java.lang.Math.*;
import static java.lang.Integer.*;
import static java.lang.Byte.*;

class C1 {
    public static void main(String[] args) {
        out.println( abs(-3) ); // System.out.println(Math.abs())
        out.println(MAX_VALUE); // Error: Byte? o Integer?
    }
}
```

AGRUPAR CLASES EN PAQUETES (directiva package)

Un programador puede crear nuevos paquetes. Si tu quieres que algunas de las clases que estás escribiendo estén dentro de un paquete llamado utilidades, entonces, tus ficheros de código fuente deben comenzar con la sentencia:

```
package nombre_del_paquete;
```

Debe aparecer antes que cualquier directiva import en el fichero y el

UNIDAD 4. Programación Modular, Clases y Objetos.

código fuente del fichero debe colocarse en una carpeta con el mismo nombre que el paquete y las clases que estén en un subpaquete deben estar en una subcarpeta. Por ejemplo: una clase que esté en un paquete llamado utilidades.net debe estar en una carpeta llamada net dentro de una carpeta llamada utilidades.

Las clases de un paquete tienen acceso automáticamente a otras clases del mismo paquete, es decir, no tienes que importarlo.

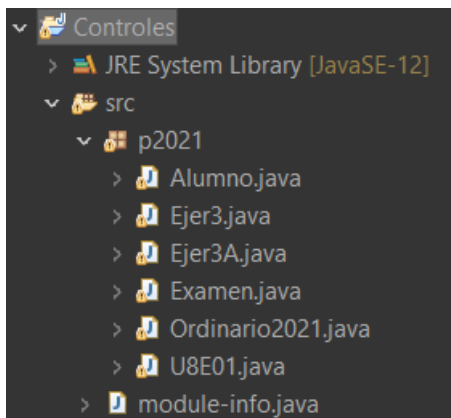


Figura 4: Proyecto donde todos los fuentes están dentro del package p2021.

En las versiones recientes de Java, los paquetes se almacenan en archivos .jar en una subcarpeta del Java Runtime Environment, como rt.jar.

Si una clase no indica de forma explícita un package, se ubica en el denominado paquete por defecto, que no tiene nombre. Por ahora no es importante, pero **cuando nuestros programas sean de varias clases, debemos asegurarnos de que estas están dentro de un paquete.** Por eso en Eclipse, antes de crear un fichero fuente en la carpeta src, debajo de la carpeta debemos crear un package y dejar nuestra clase bajo él.



UNIDAD 4. Programación Modular, Clases y Objetos.

Cuando tenemos un programa grande (con muchas clases), lo ideal es meter todos estos ficheros y directorios en un único fichero comprimido. La herramienta **jar** del JDK nos permite hacer esto. Empaqueta todo lo que le digamos (directorios, clases, ficheros de imagen o lo que queramos) en un único fichero de extensión **.jar**. Lo bueno es que puedes ejecutar el fichero directamente sin tener que descomprimirlo. En el fichero **.jar** puedes dejar los **.class** y recursos que necesite el programa y no dejar los fuentes, de forma que puedes no compartirlos con un cliente si es lo que necesitas.

EJEMPLO 20: Crear un ejecutable **.jar** desde línea de comandos. Creamos un proyecto con 2 clases **C1** (la que se ejecuta) y **C2** (usada por **C1**) en la carpeta de nombre "pami" (package **mio**).

```
// Fichero C1.java dentro de pami
package pami;

public class C1 {
    public static void main(String[] args) {
        C2 c2 = new C2();
        System.out.println( c2.saluda() );
    }
}

// Fichero C2.java dentro de pami
package pami;

public class C2 {
    public String saluda() { return "Hola, soy C2"; }
}
```

Abrimos un terminal y entramos en la carpeta **pami** y compilamos ambas clases para generar los ficheros **.class**:

```
cd .\pami
javac C1.java C2.java
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Ahora debemos crear un fichero **manifest**. Es un fichero de texto con extensión **.mf** que describe características de la aplicación que empaqueta el **.jar**, por ejemplo, qué clase es la que debe ejecutarse. Creamos el fichero **tmp.mf** con este contenido en la carpeta **pami**:

```
Main-Class: pami.C1
```

La primera línea indica que la clase ejecutable es **C1** del paquete **pami** (observa que no tiene **.java**).

Nos vamos a la carpeta padre de **pami** y creamos el **jar** y añadimos todos los ficheros (si tu programa usa imágenes, sonidos, vídeos...) y subcarpetas de la carpeta **pami** indicando el nombre de la carpeta:

```
jar -cf pami.jar pami
```

O solo los de las clases y no el resto de ficheros:

```
jar -cf pami.jar pami\*.class
```

También podemos modificar cosas, por ejemplo si tenemos que recompilar **C2**, con la opción **u** lo actualizamos (**update**):

```
jar -uf pami.jar pami\C2.class
```

Ya puedes ejecutarlo, pero debes indicar el nombre de la clase. Para ahorrarnos este trabajo hemos creado el **manifest**. Para añadirlo usamos el comando:

```
jar -umf pami/tmp.mf
```

También podríamos haberlo incluido directamente al crear el **jar**:

```
jar -cmf pami/tmp.mf pami.jar pami\*.class
```

Por último para ejecutarlo:

```
java -jar pami.jar
```



UNIDAD 4. Programación Modular, Clases y Objetos.

La ventaja de los IDES es que en aplicaciones complejas, hacer estas tareas a mano es laborioso, ellos se ocupan de este trabajo rutinario y nos podemos centrar en la parte más creativa de la programación.

4.6. CLASES Y OBJETOS.

Mientras que un método representa una tarea sencilla, un objeto encapsula tanto datos (variables de instancia) y comportamiento (en forma de métodos de instancia). Así que los objetos son otra estructura adicional que se usa para ayudar a manejar la complejidad de programas grandes.

Hay varias formas de crear objetos, una es la que usan la mayoría de los lenguajes orientados a objetos, los objetos se crean cuando se instancia una clase. La clase es por tanto una forma de describir los objetos que se pueden instanciar (crear), como una plantilla o un molde de sus objetos. Aunque también tienen otros usos. Otros lenguajes (como javascript) utilizan la técnica de prototipado, una especie de herencia.

4.6.1. DIFERENCIAS ENTRE CLASE Y OBJETO.

Si un objeto es un grupo de datos y de métodos y una clase también ¿Qué diferencia hay entre una clase y un objeto?

Las clases describen a los objetos, o de forma más exacta, sus partes no estáticas describen a los objetos. La terminología más común dice que los objetos pertenecen a una clase (la clase es como su tipo). Pero desde el punto de vista de un programador, las clases se usan principalmente para crear los objetos. Una clase es el diseño, la plantilla o el molde a partir del cual se crea un objeto. Los elementos no estáticos de la clase indican o describen qué variables y métodos contendrá cada objeto.



UNIDAD 4. Programación Modular, Clases y Objetos.

Así que la principal diferencia entre clase y objeto es que las clases se crean una vez y los objetos se crean y se destruyen durante la ejecución de un programa. Puedes crear muchos objetos a partir de la misma clase y todos comparten la misma estructura pero no tienen el mismo estado (valores de las variables de instancia, las variables que no son static).

Considera una sencilla clase que describe al usuario que está utilizando un programa:

```
class Usuario {  
    static String nombre;  
    static int edad;  
}
```

Podemos dibujar la clase en la memoria:

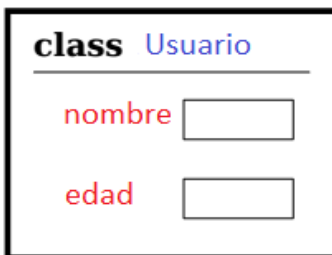


Figura 5: dibujo de una clase sencilla en memoria.

Si un programa usa esta clase, la clase solo existe una vez, por tanto solo hay una copia de las variables `Usuario.nombre` y `Usuario.edad` al tener el modificador `static`. Tendrá una zona de memoria dedicada a almacenar los valores de estas variables.

Si el programa quisiera trabajar con varios usuarios, con la clase diseñada así no podría, solo sirve para representar a un usuario. Podrías pensar en cambiar cada dato por un array y así tener varios. Pero el enfoque de la programación orientada a objetos es que cada



UNIDAD 4. Programación Modular, Clases y Objetos.

cosa del mundo real se corresponda con un objeto de un programa.

Además, la clase existe desde que el programa comienza a funcionar hasta que acaba de ejecutarse. Lo ideal sería que un usuario exista desde que se necesita trabajar con él hasta que ya no se necesite.

Ahora, vamos rediseñar otra clase similar para que tenga variables miembro no estáticas (cada objeto tendrá sus propios valores):

```
class Jugador {  
    static int numJugadores; // Solo un valor en la clase  
    String nombre;           // Cada objeto tiene un valor nombre  
    int edad;                // Cada objeto tiene un valor edad  
}
```

Si creamos 3 objetos Jugador en un programa, tendremos en RAM algo similar a la figura 6.

Ahora esta clase solamente tiene una variable estática (de la clase). Los datos nombre y edad ya no son de la clase (por eso no podemos referenciarlos como Jugador.nombre y Jugador.edad). Estas variables no estáticas son de cada objeto que se cree a partir de la clase, cada uno debe tener su propia memoria para almacenar estos valores. Así que **la parte no estática de una clase es una plantilla usada para crear sus objetos.**

Un objeto creado a partir de una clase se denomina **una instancia de la clase** que significa una existencia real de la plantilla (se dibuja con una flecha que sale del objeto y apunta a su clase). En la figura, la clase contiene algo denominado **constructor** que indica la existencia en la clase de un método encargado de crear los objetos (a estos métodos especiales, una clase puede tener varios, se les llama **constructores**).



UNIDAD 4. Programación Modular, Clases y Objetos.

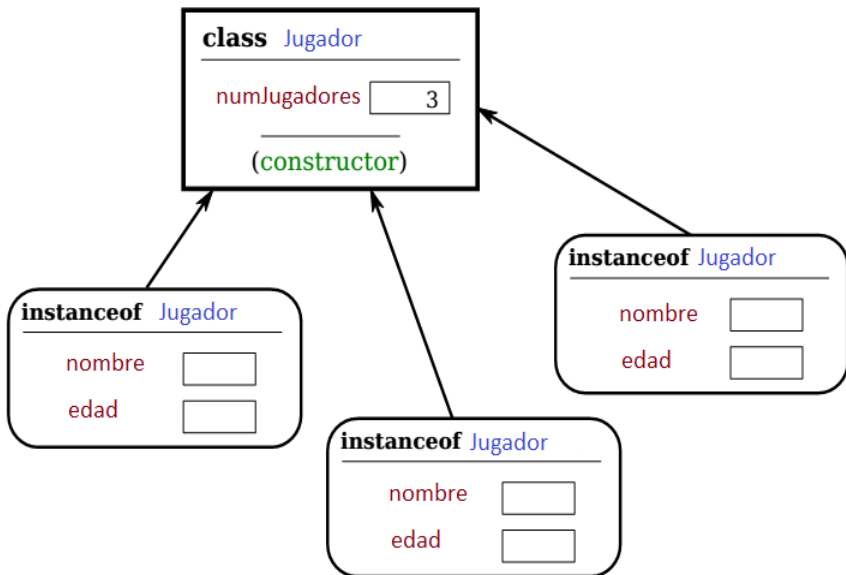


Figura 6: dibujo de una clase y 3 objetos suyos instanciados en memoria.

Las variables que son de cada objeto se llaman **variables de instancia** y los métodos que son de los objetos se llaman **métodos de instancia**. Aunque el ejemplo no incluye ningún método, la mecánica es idéntica: si tiene el modificador **static** es de la clase, sino, es de los objetos.

En la vida real se dan todos los casos: verás clases que solo tienen miembros estáticos, mezclados o solamente miembros de instancia.

Vamos a comenzar con una versión simplificada de una clase que represente a un estudiante de un curso:

```
public class Alumno {
    public String nombre;           // Nombre del alumno
    public double eval1, eval2, eval3; // Notas de las evaluaciones
    public double getMedia() {      // calcula la nota media
        return (eval1 + eval2 + eval3) / 3;
    }
} // fin Alumno
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Ninguno de los miembros es estático, así que esta clase solo sirve para crear objetos. **En Java, una clase es como un tipo de dato.** Así que podemos usarlo en la sentencia de declaración de variables, o como tipo de un parámetro formal en un método, o el dato devuelto en una función con la sentencia **return**.

```
Alumno a1;  
public void enviarNotas(Alumno a) { ... }
```

En el caso de los objetos, declarar una variable no crea el objeto: en Java, una variable no almacena un objeto, solamente lo referencia (señala o apunta a la zona de memoria donde está almacenado). Los objetos se almacenan en una zona de memoria llamada **heap**. La variable tiene información para localizarlo, pero no lo almacena.

En un programa los objetos se crean usando el operador `new`, que crea el objeto y devuelve una referencia a la memoria donde existe. El operador **`new`** llama a un método especial de la clase llamado **constructor**. Ej:

```
a1 = new Alumno();
```

Como la variable `a1` referencia a un objeto o instancia de la clase `Alumno`, para acceder a sus datos y métodos debemos usar la variable que los referencia y el nombre de las variables y métodos, por ejemplo:

```
System.out.println("Hola, " + a1.nombre + ". Tus notas son:");  
System.out.println(a1.eval1);  
System.out.println(a1.eval2);  
System.out.println(a1.eval3);
```

Lo mismo se aplica a los métodos. Puedes usar los datos en expresiones según el tipo que tengan. Ej:

```
System.out.println( "Tu media es " + a1.getMedia() );  
a1.nombre.length()
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Cuando una variable de tipo clase, no referencia a ningún objeto, decimos que es una referencia nula, valor **null**. Puedes almacenar null en una variable de tipo referencia:

```
a1 = null;
```

Para saber si una variable está referenciando a un objeto, puedes comprobarlo. Ej:

```
if( a1 == null ) ...
```

Nota: si intentas acceder a un miembro de un objeto no creado (referencia nula), se lanza una excepción **NullPointerException**.

Vamos a ver una secuencia de sentencias que trabajan con objetos:

```
Alumno a, a1, a2, a3; // Declara 4 variables de tipo Alumno
a = new Alumno();     // Crea un objeto y guarda la referencia
a1 = new Alumno();    // Crea un segundo objeto Alumno
a2 = a1;              // Copia la referencia en a2
a3 = null;            // a3 no referencia nada
a.nombre = "John Smith"; // guarda un nombre en el objeto a1
a1.nombre = "Mary Jones";
// El resto de variables de instancia tiene valor por defecto 0.0
```

Lo que estas sentencias consiguen, representado en memoria sería lo que aparece en la figura 7. Las referencias se han representado mediante flechas. Observa que los String son objetos, por tanto las variables también son referencias. Pero lo interesante de esta figura es lo que ocurre en a1 y en a2: **cuando una variable objeto se asigna a otra, se copia la referencia pero no se hace una copia del objeto.**



UNIDAD 4. Programación Modular, Clases y Objetos.

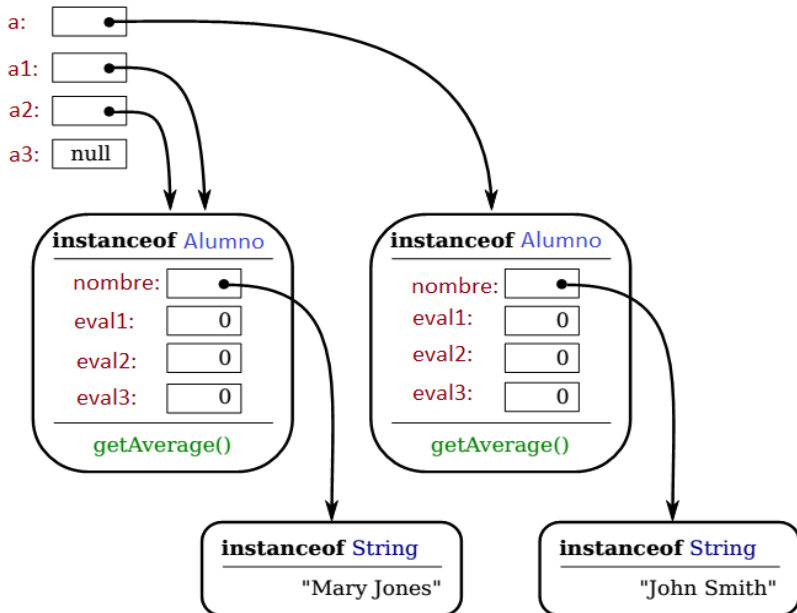


Figura 7: aspecto en memoria de variables referenciando objetos.

Las dos variables acaban apuntando al mismo objeto, no se ha creado un nuevo objeto copiando/clonando al original, es el mismo objeto. Por tanto si cambias algo del objeto de `a1`, estás cambiando también el objeto de `a2`, porque en realidad, son el mismo objeto.

Puedes comprobar la igualdad o desigualdad de dos objetos usando el operador `==` y `!=`, pero de nuevo debes comprender bien lo que haces, **lo que comparas son las referencias, no los objetos**, comparas si estás refenciando al mismo objeto, no estás comprobando si los objetos que referencias tienen los mismos valores. Para comprobar si los objetos `a1` y `a2` tienen los mismos valores debes hacer:

```
a1.eval1 == a2.eval1 && a1.eval2 == a2.eval2 &&
a1.eval3 == a2.eval3 && a1.nombre.equals(a2.nombre)
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    String s1 = "Hola";  
    System.out.print("Me saluda? ");  
    String s2 = sc.next();  
    if( s1 == s2 ) {  
        System.out.println("Son iguales");  
    } else {  
        System.out.println("No son iguales"); // el resultado!!  
    }  
}
```

Observe que aunque los string son objetos, aunque lo único que almacenan es la cadena de caracteres, si usas los comparadores de igualdad (==) y desigualdad (!=) puedes tener problemas, porque no estás comparando las cadenas si no las referencias a esas cadenas.

```
Console x Declaration  
<terminated> U8E01 [Java Applic  
Me saluda? Hola  
No son iguales
```

Imagine que una variable que referencia a un objeto se declara como **final**. Eso significa que el valor almacenado nunca puede ser cambiado tras inicializar la variable. Esto significa que siempre referencia al mismo objeto, pero no impide que el objeto cambie. Ej:

```
final Alumno A = new Alumno(); // La referencia es constante  
A.nombre = "John Doe";        // Cambia el objeto, no la referencia
```

Ahora, imaginemos que la variable **obj** la pasamos a un método (obj referencia a un objeto), ya sabemos que el método no puede cambiar el valor de la variable, puesto que trabaja con una copia. Pero, la copia es de la referencia, no del objeto. Luego si cambias el objeto dentro del método, los cambios son visibles en el exterior. Ej:

```
void noCambia(int z) {  
    z = 42;  
  
void cambia(Alumno a) {  
    a.nombre = "Fred";
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
}
```

```
}
```

Las llamadas:

```
x = 17;  
a.nombre = "Jane";  
noCambia(x);  
cambia(a);  
System.out.println(x);  
System.out.println(a.nombre);
```

Salida: 17 y "Fred".

PALABRA RESERVADA **this**

La palabra reservada **this** representa al objeto actual que está ejecutando el código donde aparece.

Como representa al objeto actual que ejecuta el código, **no se puede usar desde dentro de un bloque de sentencias estático** (de la clase) donde **this** (el objeto) no tiene sentido.

Se utiliza para usar algo del objeto actual (una variable, un constructor, etc.) y quitar ambigüedad en ciertas circunstancias y para llamar a un constructor desde otro o para pasar el propio objeto a otro método de la misma o diferente clase.

```
// Usarlo para quitar ambigüedad en un método  
// variable miembro llamada nombre guardo el parámetro nombre  
public void setNombre(String nombre) { this.nombre = nombre; }  
// Usarlo para pasar el propio objeto a otro método de  
// la misma o de otra clase (de otra en el ejemplo)  
Pantalla.dibuja(this, 5, 6);  
// Llamar a otro constructor desde otro constructor  
// La clase C1 debe tener otro constructor definido como  
// public C1(int parametro) {...} por ejemplo, y la llamada  
// debe ser la primera sentencia en aparecer en el constructor  
public C1(){ this(3); }
```

4.6.2. GETTERS Y SETTERS



UNIDAD 4. Programación Modular, Clases y Objetos.

Cuando se crea una nueva clase hay que prestar atención al acceso que se permite desde el exterior a sus variables miembro. El código interno a la clase siempre puede acceder a las variables globales (salvo desde el código estático que solo accede a variables estáticas). Pero la cosa cambia si se intenta acceder desde el exterior.

Java permite cuatro posibilidades de control de acceso desde el exterior: **public** significa que se tiene acceso desde cualquier lugar; **private** restringe el acceso a los métodos de la clase; **no indicar nada (o poner package)** permite acceso desde cualquier otra clase del mismo paquete. Y **protected** que permite el acceso desde las clases hijas y desde el package.

La **encapsulación** (agrupar cosas) y la **ocultación** de detalles son principios de la orientación a objetos. La mayoría de las variables de una clase deberían ser privadas. Y para permitir que desde fuera se puedan leer/modificar, la propia clase implementa métodos que hacen posible estas operaciones. Por ejemplo, si una clase tiene una variable privada de tipo String y de nombre **titulo**, puedes crear este método para permitir leerla:

```
public String getTitulo() { return titulo; }
```

A estos métodos se les llama **getters** y su nombre suele construirse con la palabra "get" seguida del nombre de la variable capitalizado, en el ejemplo "get" + "Titulo" -> getTitulo().

Si quieres permitir la operación de escritura (cambiar el valor), puedes implementar un método que lo haga. A estos métodos se les llama **setters** y su nombre se construye con la palabra "set" y el nombre de la variable capitalizado. En el ejemplo:

```
public void setTitulo( String nuevoTitulo ) {  
    titulo = nuevoTitulo;  
}
```




UNIDAD 4. Programación Modular, Clases y Objetos.

```
}
```

A veces, a las variables de tipo boolean, se les suele proporcionar getters que no comienzan con "get" si no con "is". Por ejemplo una variable privada booleana que se llame cobrada. Su getter podría ser: `isCobrada()` (está cobrada).

La ventaja de implementar getters y setters en vez de declarar las variables públicas es que en los métodos, además de permitir leer/escribir los datos, también puedes hacer comprobaciones, por ejemplo contar cuantas veces ha sido modificada una variable, o la fecha y hora, o quien accede, o realizar cualquier comprobación que averigue si la operación de lectura/escritura se debe permitir. Por ejemplo:

```
public String getTitulo() {  
    leenTitulo++;  
    return titulo;  
}
```

y un setter puede hacer verificaciones sobre la legalidad del valor a cambiar:

```
public void setTitulo( String nuevoTitulo ) {  
    if ( nuevoTitulo == null || nuevoTitulo.isEmpty() )  
        titulo = "(Sin título)"; // un valor por defecto  
    else  
        titulo = nuevoTitulo;  
}
```

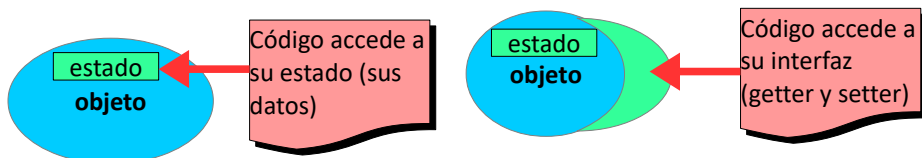
Incluso **pueden ayudarte a no tener que cambiar código si en tus clases los usas desde el principio**. Imagina que haces una clase con dos variables, usas la clase durante años en tus programas, que acceden a sus variables no directamente si no a través de los getters y setters (los programas no acceden, no conocen, no dependen del interior de la clase, has ocultado su implementación). Al cabo de dos



UNIDAD 4. Programación Modular, Clases y Objetos.

años decides (por un buen motivo) cambiar las variables que contiene. ¿Eso implica que debes modificar todos los programas que la han usado durante esos años? No, simplemente debes mantener sus getters y setters. En ese caso, **los cambios se aíslan de forma eficiente**.

Getters y setters **definen una propiedad de la clase** que podría corresponderse internamente con una, varias o ninguna variable en realidad. Eso queda oculto al exterior (encapsulas y ocultas el interior). Esto hace que tu código **baje el acoplamiento** (las interdependencias de unos elementos con otros) lo que aumenta la calidad al aumentar la facilidad de mantenimiento y reutilización.



a) El código se hace dependiente de la implementación del objeto

b) El código es independiente de la implementación

Figura 8: a) Acceso directo -> alto Acoplamiento (no usas ocultación).

b) Con getters y setters creas una interfaz del estado -> independizas implementación, bajas acoplamiento y facilitas comprobaciones.

4.6.3 ARRAYS Y STRINGS SON OBJETOS.

Ya hemos trabajado con Strings (los operadores de igualdad comprueban si apuntan al mismo objeto, no si sus caracteres coinciden, etc.) y arrays, que también son objetos y se les aplica las mismas propiedades que a cualquier objeto, así que:

- Una variable array es solo una referencia a la memoria heap donde se almacena el objeto con sus valores.
- Nunca es posible pasar un array por valor (copia) cuando lo usas como parámetro actual.
- Si asignas un array/string a otra variable, no copias los



UNIDAD 4. Programación Modular, Clases y Objetos.

elementos, simplemente se copia la referencia (las variables apuntan al mismo objeto).

- Si accedes a un elemento o a la propiedad **length** de un array que no referencia a nada, se genera la excepción **NullPointerException**.
- Para saber si un array no referencia, puedes comparar con null.
- Si comparas dos arrays con el operador de igualdad, no estás comparando sus elementos, sino si son el mismo objeto.

Ejemplos:

```
int[] lista = { 1, 2, 3};
int[] nuevalista;
nuevalista = lista; // Las dos apuntan al mismo objeto
nuevalista[1] = 0;
System.out.println( lista[1] ); // Imprime 0
```

Los arrays son objetos que pueden almacenar objetos, es decir su tipo base puede ser una clase. Podemos tener arrays de cadenas `String[]` o de cualquier otra clase. Podemos tener por ejemplo un array de la clase `Alumno` que definimos antes: `Alumno[]`

```
Alumno[] aula; // Declara una variable array de alumnos
aula = new Alumno[30]; // ahora referencia un objeto array
```

Cuando se crea el array de objetos, los elementos se inicializan con el valor por defecto de los objetos que es null, así que `aula[0] ... aula[29]` tienen null. Para rellenar los elementos, habrá que crear 30 objetos de la clase `Alumno` (instanciar la clase 30 veces):

```
Alumno[] aula;
aula = new Alumno[30];
for ( int i = 0; i < 30; i++ ) {
    aula[i] = new Alumno();
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Para acceder al alumno de la posición 3 podemos usar `aula[3]` que es a todos los efectos un objeto de la clase `Alumno`:

```
aula[3].nombre; // nombre del Alumno de la posición 3 del array
```

4.7. CONSTRUCTORES E INICIALIZACIÓN.

Los objetos, al contrario que los tipos de datos primitivos, además de declararse, para existir deben crearse de forma explícita. La creación implica buscar memoria en el heap para almacenarlo, y dar un primer valor a sus variables de instancia. La memoria donde se quede almacenado, en principio te da igual, pero si te interesa en muchas ocasiones controlar qué valores se les da a las variables globales.

4.7.1 INICIALIZAR VARIABLES GLOBALES.

Al declarar una variable global (estática o no) se le da un valor por defecto. También se puede asignar un valor inicial al declararla como a cualquier otra variable. Igual ocurre con las variables estáticas, que se pueden inicializar al cargar la clase.

Salvo declaraciones e inicializaciones no puede haber sentencias que no estén en un bloque, podríamos tener esto:

```
public class Lanza2Dados {  
    static public numLanzamientos = 0; // Al cargar la clase  
    public int dado1 = 3; // Nº del primer dado, al crear objeto  
    public int dado2;      // Nº del segundo dado, a 0 por defecto  
    static { numLanzamientos++; } // Al cargar la clase  
    { dado2 = 4; }              // Al crear cada objeto  
}
```

Ahora hacemos una nueva versión de `Lanza2Dados` que representa el lanzamiento de dos dados. Tiene dos variables de instancia que contienen los números actuales de los dados (`dado1` y `dado2`) y un método `lanza()` que simula su lanzamiento:



UNIDAD 4. Programación Modular, Clases y Objetos.

```
public class Lanza2Dados {  
    public int dado1 = 3; // Nº del primer dado  
    public int dado2 = 4; // Nº del segundo dado  
    public void lanza() {  
        dado1 = (int)(Math.random() * 6) + 1;  
        dado2 = (int)(Math.random() * 6) + 1;  
    }  
} // fin clase
```

La inicialización de las variables de instancia se ejecuta cuando el objeto es creado, y queda con dado1 a 3 y dado2 a 4. Una variación de la clase:

```
public class Lanza2Dados {  
    public int dado1 = (int)(Math.random() * 6) + 1;  
    public int dado2 = (int)(Math.random() * 6) + 1;  
    public void lanza() {  
        dado1 = (int)(Math.random()*6) + 1;  
        dado2 = (int)(Math.random()*6) + 1;  
    }  
} // fin clase
```

Ahora cada nuevo objeto probablemente tenga valores diferentes a otro, al asignarlos de forma aleatoria. Si no das un valor inicial a las variables, se les da el valor por defecto del tipo que tengan (0 para tipos numéricos, el carácter UNICODE 0 para char, false para booleanos y null para objetos).

4.7.2 CONSTRUCTORES.

Ya sabes que los objetos se crean con el operador **new**. Por ejemplo:

```
Lanza2dados dado; // Declara una variable  
dado = new Lanza2Dados(); // Construye objeto y da la referencia
```

En este ejemplo, la expresión **new Lanza2dados()** le busca memoria al objeto, inicializa sus variables de instancia y devuelve la referencia al nuevo objeto. La referencia se guarda en la variable dado.

El método **Lanza2Dados()** se dice que es un **constructor**. Existe aunque



UNIDAD 4. Programación Modular, Clases y Objetos.

nadie lo haya definido, es decir, **toda clase tiene como mínimo un constructor, si el programador no lo define, se crea uno de forma automática (el constructor por defecto)**. El constructor por defecto solo hace cosas básicas (aloja memoria e inicializa las variables a sus valores por defecto). Una clase puede definir más de un método constructor.

Nota: debes tener muy presente que cuando se ejecuta el código que escribes dentro de un constructor, es porque en algún lugar se ha usado una sentencia `new Constructor(...)`;

Un constructor es como cualquier otro método salvo por 3 cosas:

1. No devuelve ningún dato, ni siquiera **void**.
2. El nombre siempre coincide con el de la clase.
3. Solo puede tener modificadores de acceso (**public, private y protected**).

Todo lo demás lo comparte con los métodos: tiene un bloque de sentencias y parámetros formales. Los parámetros se usan para pasar información que se utiliza para construir el objeto. Por ejemplo podemos crear uno para que defina el valor inicial de los dados:

```
public class Lanza2dados {  
    public int dado1; // Nº del primer dado  
    public int dado2; // Nº del segundo dado  
    // Constructor: Crea el objeto y da valores de parámetros  
    public Lanza2Dados(int valor1, int valor2) {  
        dado1 = valor1;  
        dado2 = valor2;  
    }  
    public void lanza() {  
        dado1 = (int)(Math.random() * 6) + 1;  
        dado2 = (int)(Math.random() * 6) + 1;  
    }  
} // fin clase
```



UNIDAD 4. Programación Modular, Clases y Objetos.

El constructor que hemos creado tiene 2 parámetros así que ahora podemos crear objetos de esta forma:

```
Lanza2Dados dados;           // Declarar variable
dados = new Lanza2Dados(1,1); // se crea con valores 1,1
```

Pero ahora que hemos definido un constructor de manera explícita, el constructor por defecto lo perdemos (**solo tenemos constructor por defecto si no hay definido ninguno**), si queremos tener un constructor por defecto (sin parámetros) además de otros, hay que definirlo de forma explícita. Añadimos a la clase este otro:

```
public Lanza2Dados() {
    lanza(); // llama al método para generar valores
}
```

Cuando implementas una clase con la finalidad de usarla como plantilla para crear objetos, una vez escrita y probada, debe poder utilizarse en cualquier programa que necesites. Debe ser capaz de interactuar convenientemente con el resto de objetos que puedan existir en un programa Java y así habrás obtenido un elemento reutilizable en cualquier programa.

Para conseguir que nuestras clases sean de calidad (tengan la característica de poder reutilizarlas) deben incorporar como mínimo ciertos elementos hasta conseguir hacerlas similares a una **clase canónica** (una clase con los elementos mínimos necesarios para tener un buen comportamiento en el entorno donde se utiliza). Estos elementos los examinaremos más adelante.

```
public class Lanzamientos {

    public static void main(String[] args) {
        Lanza2Dados primero;           // Primer par de dados
        primero = new Lanza2dados();
        Lanza2dados segundo;           // Segundo juego de dados
        segundo = new Lanza2Dados();
    }
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
int tiradas;                // Cuenta los lanzamientos
int total1;                 // Total del primer juego
int total2;                 // Total del segundo juego
tiradas = 0;
do {                        // lanzar hasta que los totales coincidan
    primero.lanza();
    total1 = primero.dado1 + primero.dado2;
    System.out.println( "Primero sale " + total1 );
    segundo.lanza();
    total2 = segundo.dado1 +segundo.dado2;
    System.out.println("Segundo sale " + total2 );
    lanzamientos++;        // Contar este lanzamiento
    System.out.println();
} while (total1 != total2);
System.out.println("Coinciden en " + lanzamientos
    + " lanzamientos.");
} // fin main()
} // fin clase
```

LLAMADA A UN CONSTRUCTOR DESDE OTRO

A veces necesitaremos 4 o 5 constructores diferentes para una misma clase (sobre todo si es compleja). Para simplificar el código, es interesante hacer que uno de ellos recoja mucha de la complejidad de la construcción (comprobaciones, inicializaciones, etc.), y el resto de constructores se simplifican porque lo usan haciendo una llamada.

Los constructores son métodos especiales: no son métodos de instancia (de los objetos) porque no hay objeto cuando se utilizan por primera vez (crearlos es su trabajo). Son más parecidos a los métodos de clase, pero no pueden ser estáticos, así que no son de la clase. Por eso al final no se les llama métodos. Además, solo se pueden llamar junto con el operador **new** siguiendo la sintaxis:

new Clase(lista_parámetros);

La llamada a un constructor es más complicada que a un método normal. Veamos los pasos que realiza:



UNIDAD 4. Programación Modular, Clases y Objetos.

1. Obtiene un bloque de memoria sin usar del heap, lo suficientemente grande para almacenar al objeto.
2. Inicializa las variables de instancia, si la declaración indica un valor se usa, si no, el valor por defecto.
3. Los parámetros actuales del constructor (si los hay) se evalúan y asignan a los parámetros formales.
4. Se ejecutan las sentencias del cuerpo del constructor (si las hay).
5. Se devuelve una referencia al objeto creado.

IMPORTANTE: Si desde un constructor queremos llamar a otro constructor para aprovechar su código:

- Debemos llamar a uno que exista (casen los parámetros).
- Dentro de un constructor de la clase C1, no sirve de nada usar `new C1()` porque no podremos devolver el objeto recién creado. Dentro del constructor de C1 ya estamos dentro de una operación `new C1()`, es decir, podríamos crear un bucle de llamadas recursivas: `public C1() { C1 c = new C1(); }`
- Debe ser la primera sentencia que aparezca.
- No podemos usar el nombre de la clase, sino **`this(...)`**:

EJEMPLO 21: Redefine clase Alumno y comenta varios trucos/mejoras con el uso de constructores y modificadores de acceso.

```
public class Alumno {
    private final String nombre;           // Nombre del alumno
    public double eval1, eval2, eval3;     // Notas en evaluaciones
    // Constructor con nombre -> Impide alumnos con nombre a null
    public Alumno(String suNombre) throws IllegalArgumentException{
        if ( suNombre == null || suNombre.isEmpty() )
            throw new IllegalArgumentException("Debo tener nombre");
        nombre = suNombre;
    }
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
public String getNombre() { // Getter del nombre
    return nombre;
}
public double getMedia() { // Getter de un dato calculado
    return (eval1 + eval2 + eval3) / 3;
}
} // fin de clase
```

Primera mejora: No deja que se creen alumnos sin nombre, no hay otra forma de crearlos (obligamos al programador a dar un valor de una propiedad). El constructor en esta clase tiene un parámetro de tipo String, que indica el nombre de estudiante.

Segunda mejora: al usar el modificador private, impedimos que se pueda cambiar el nombre de un alumno y ni siquiera consultarlo. Para dejar leerlo, definimos un getter, pero no dejamos que se cambie una vez asignado.

Nota: *es un buen estilo de programación declarar un dato que no queremos que cambie como **final**. El único lugar donde está permitido darle un valor es dentro de un constructor, porque se considera el primer valor.*

EJEMPLO 22: Sacar partido de tener datos estáticos (de clase) y de instancia (de los objetos). Necesitamos que cada nuevo alumno tenga un código que lo diferencie del resto de alumnos que haya en el programa, es decir, que no se repita en dos alumnos diferentes (como un NIA). ¿Cómo programar eso? Vamos a conseguirlo teniendo ese código NIA como un dato de la clase, cada nuevo objeto que se cree, incrementamos el NIA siguiente a usar.

```
public class Alumno {
    private String nombre; // Nombre del alumno
    public double eval1, eval2, eval3; // Notas en evaluaciones
    private int NIA; // Único para cada alumno
    private static int siguienteNIA = 0;
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
// Constructor con nombre -> Impide alumnos con nombre a null
Alumno(String suNombre) {
    if ( suNombre == null )
        throw new IllegalArgumentException("Debo tener nombre");
    nombre = suNombre;
    siguienteNIA++;
    ID = siguienteNIA;
}
public String getNombre() { // Getter del nombre
    return nombre;
}
public int getNIA() { // Getter del NIA
    return NIA;
}
public double getAverage() {
    return (eval1 + eval2 + eval3) / 3;
}
} // fin de clase
```

Nota: en programas que usen más de un thread (varios hilos y cada hilo creando en paralelo alumnos) no hay garantías de obtener un NIA único, habría que crear zonas de exclusión mutua en el código, pero eso se sale del presupuesto...

4.7.3 RECOLECCIÓN DE BASURA.

Un objeto existe en el heap y puede ser accedido a través de las variables que almacenan su referencia. ¿Qué hacer con un objeto del que no se guarda su referencia? Considera estas sentencias:

```
Alumno a1 = new Alumno("Juanita Pérez");
a1 = null;
```

La segunda sentencia hace que el programa ya no pueda trabajar con el objeto que crea la primera sentencia. La memoria que consume el objeto ya no es útil y si se libera, otros objetos la pueden aprovechar. Pero Java (al contrario que otros lenguajes como C++) no tiene sentencias para devolver la memoria que ocupan los objetos.



UNIDAD 4. Programación Modular, Clases y Objetos.

Java usa un procedimiento llamado **garbage collection** (recolección de basura) para recuperar la memoria ocupada por objetos que ya no son accesibles para el programa. Este trabajo ha pasado a ser responsabilidad del sistema no del programador. Si un objeto tiene varias referencias, y una se pierde, aún puede ser utilizado por el resto. Por eso, un objeto no es candidato a ser basura mientras haya referencias que lo puedan usar.

Nota: *el que los programadores sean responsables del consumo y liberación de la memoria, ha sido una gran fuente de errores y pérdida de recursos en los programas.*

La idea de la recolección de basura viene de los años 1960, pero no se implementó en lenguajes anteriores porque necesita consumir recursos para automatizarla. Pero Java surgió cuando el hardware ya tenía suficiente capacidad para implementar esta característica (para regocijo y alivio de los probrecitos programadores de todo el mundo).

4.8 IMPLEMENTAR CORRECTAMENTE CLASES.

En Java se denomina **clase canónica** a aquella que contiene los elementos necesarios para poder utilizarla en cualquier otro programa con facilidad. Por ejemplo: modificadores de acceso a sus variables, constructores adecuados, getters y setters que ayuden a intercambiar información con el exterior además de mantener el estado del objeto consistente, métodos `toString()`, `equals()`, `compareTo()`, `hashCode()`,...

En este apartado vamos a comenzar a revisar como implementar buenas clases. Dejaremos para temas posteriores algunos de los métodos que deberían tener nuestras clases, a medida que lo vayamos necesitando. Por ahora nos centraremos en `toString()` y `equals()`.



UNIDAD 4. Programación Modular, Clases y Objetos.

4.8.1 LA CLASE OBJECT

Ya hemos comentado que una de las mayores virtudes de la POO es poder crear nuevas clases a partir de otras ya existentes. Las subclases (las nuevas) heredan todas las propiedades y el comportamiento de la clase original, pero además pueden modificar las características existentes o añadir nuevas propiedades y métodos.

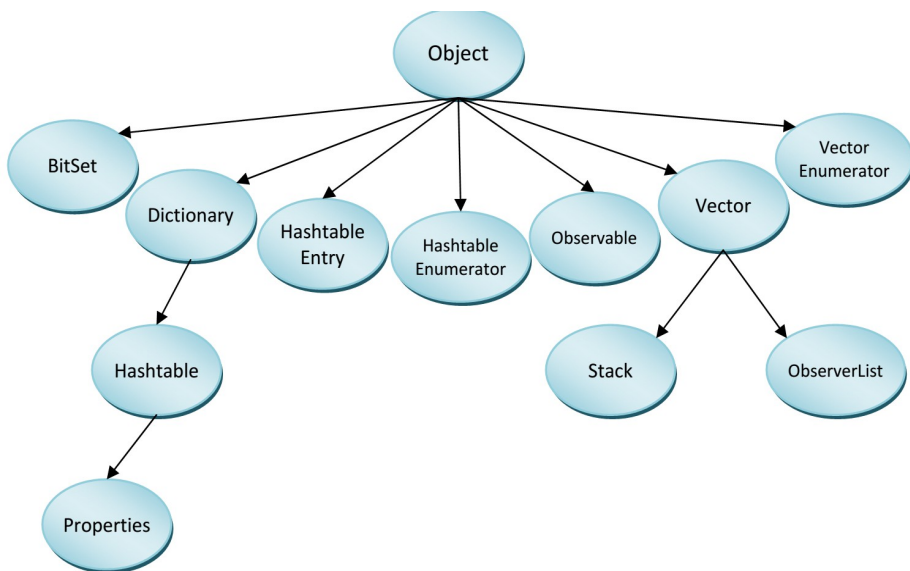


Figura 9: Object está en la raíz de las Jerarquías de clases Java.

En Java todas las clases son subclases de alguna otra, con una única excepción, **la clase Object**. Si creas una nueva clase y no la extiendes de ninguna otra (no indicas que es hija de otra clase), automáticamente se convierte en subclase de la clase **Object**.

Object es la única clase en Java que no es subclase de otra, **es el Adán y la Eva de las clases en Java**. El objeto padre de todos los objetos de un programa Java.



UNIDAD 4. Programación Modular, Clases y Objetos.

La clase `Object` define varios métodos de instancia que son heredados por todo el resto de objetos. Por ejemplo, presta el constructor por defecto a cualquier objeto que no defina sus propios constructores, presta el método de instancia `toString()` que devuelve un `String` con la representación del objeto. Ya lo has usado sin saberlo cada vez que querías imprimir algún dato no `String` por pantalla, el objeto se convertía en `String` gracias a este método.

En la siguiente tabla aparecen algunos de los métodos que otros objetos heredan de la clase `Object` y en la mayoría de las ocasiones es conveniente sobreescribirlos (sustituirlos) siempre (`equals` y `toString`) y otros según se necesite (`clone`, `hashCode`, etc.).

MÉTODO	DESCRIPCIÓN	SE SOBREScribe
<code>Object clone()</code>	Crea y devuelve una copia ligera del objeto	no
<code>boolean equals(Object o)</code>	true si los objetos son el mismo (<code>this == o</code>)	si
<code>void finalize()</code>	Lo llama el recolector de basura si detecta que no hay más referencias al objeto.	no
<code>Class<?> getClass()</code>	la clase en tiempo de ejecución del objeto	no
<code>int hashCode()</code>	un hash del valor del objeto	sí
<code>String toString()</code>	representación en <code>String</code> del objeto. Es recomendable que todas las clases lo sobreescriban. Por defecto devuelve: <code>getClass().getName() + '@' + Integer.toHexString(hashCode())</code>	sí



UNIDAD 4. Programación Modular, Clases y Objetos.

MÉTODO	DESCRIPCIÓN	SE SOBREScribe
<code>void notify()</code>	Si hay threads bloqueadas esperando a poder usar el objeto, una de ellas se desbloquea.	no
<code>void notifyAll()</code>	Desbloquea todos los threads bloqueados en el objeto.	no
<code>void wait()</code>	El thread actual se bloquea en el objeto	no
<code>void wait(ms)</code>	El thread actual se bloquea en el objeto durante ms milisegundos como máximo	no
<code>void wait(ms, ns)</code>	El thread se bloquea como máximo ms milisegundos y ns nanosegundos	no

MÉTODO `toString()`

Todo objeto Java debería ser capaz de mostrarse en formato de texto. Si una clase no lo implementa, ni tampoco sus clases padre (de las que descende), usará la versión de `Object` (el padre de todos los objetos) que se lo presta.

La versión del método implementada en la clase `Object` devuelve el nombre de la clase a la que pertenece el objeto concatenado a un carácter arroba "@" y a un número hexadecimal que es el código hash del objeto (algo muy genérico y poco descriptivo en realidad).

```
public String toString() {  
    return getClass().getName()+"@"+Integer.toHexString(hashCode());  
}
```

Por tanto, cuando crees una nueva clase en Java es conveniente que implemente su propio método **`toString()`** y sobrescriba (sustituya) el de la clase `Object`. Eso permite mostrar por pantalla al objeto. Por



UNIDAD 4. Programación Modular, Clases y Objetos.

ejemplo, vamos a crear el método para la clase Lanza2Dados, que al no extender a otra es subclase de Object:

```
/* Devuelve la representación en texto del lanzamiento
 * mostrando el valor de los dados */
@Override
public String toString() {
    if (dado1 == dado2)
        return "dobles " + dado1;
    else
        return dado1 + " y " + dado2;
}
```

Así, cuando un objeto dados necesite convertirse a String, se usará su método toString() automáticamente:

```
Lanza2Dados dados = new Lanza2dados();
System.out.println( "En los dados salen " + dados );
```

MÉTODO equals()

Ya sabemos que utilizar el operador "==" o "!=" para comparar objetos no da el resultado esperado (o1 == o2 compara las referencias no los objetos, por tanto comprueba cuando son el mismo objeto, comparten la misma memoria física) y normalmente nosotros lo que necesitaremos saber es si contienen la misma información aunque esté almacenada en distintos lugares.

Por ejemplo, para nosotros dos variables que referencien fechas (objetos Date), son iguales cuando describen el mismo momento de tiempo, mientras que para los operadores relacionales de Java solo son iguales si referencian al mismo objeto.

En la clase **Object** se define el método **equals(Object)** donde se implementa el código que hace nuestra versión de la igualdad entre los objetos obj1 y obj2. Pero como casi siempre, la implementación que tiene es demasiado genérica y casi nunca coincide con nuestras



UNIDAD 4. Programación Modular, Clases y Objetos.

necesidades, así que deberíamos sobrescribir este método en nuestras clases si queremos implementarlas de forma correcta.

EJEMPLO 23: Definir el método para poder saber si dos cartas de una baraja son iguales. Queremos que una carta sea igual a otro objeto si el otro objeto es él mismo o si es una carta y tiene le mismo palo y valor.

```
public class Carta {
    private int palo;    // espadas, copas, oros, bastos
    private int valor;   // Nº de 1 a 11

    @Override
    public boolean equals(Object obj) {
        if( obj == null ) return false;
        if( this == obj ) return true;   // Como operador ==
        if( obj instanceof Carta ) {
            Carta otra = (Carta)obj; // casting seguro por el if
            if(palo == otra.palo && valor == otra.valor) return true;
        }
        return false;
    }
    // resto de la clase...
}
```

Según la especificación del método equals definido en la clase *Object*, tus método `equals()` deben cumplir las siguientes propiedades:

- **Reflexiva:** `x.equals(x)` es true.
- **Simétrica:** `x.equals(y)` es true si `y.equals(x)` lo es.
- **Transitiva:** si `x.equals(y)` es true y además `y.equals(z)` es true, entonces `x.equals(z)` debe ser true.
- **Consistente:** varias llamadas a `x.equals(y)` siempre son true o false si ni `x` ni `y` cambian. El resultado es invariante en el tiempo.
- **`x.equals(null)`** siempre es false.



UNIDAD 4. Programación Modular, Clases y Objetos.

4.8.2 MÁS SOBRE CLASES DE USO HABITUAL.

Hasta ahora nos hemos centrado en el diseño e implementación de nuevas clases. Pero no hay que olvidar que Java tiene un gran número de clases listas para usarlas. Algunas pueden utilizarse para crear objetos directamente y otras para extenderlas (crear nuevas clases a partir de ellas). Un programador experto debe estar familiarizado con las clases preconstruidas y sacarles partido. Repasemos las que ya conocemos y veamos algunas más de las que hemos usado hasta ahora.

System

La clase `System` tiene información (**property**) que puedes pedir sobre el sistema, te permite acabar un programa (**exit**) o ejecutar un programa externo desde el actual. También tiene el método **arrayCopy** relacionado con los arrays, que permite copiar un array en otro. Recibe cinco argumentos: el array que se copia, el índice desde que se empieza a copiar en el origen, el array destino, el índice desde el que se copia en el destino, y el tamaño de la copia (número de elementos de la copia). Ejemplo:

```
int uno[] = {1, 1, 2};
int dos[] = {3, 3, 3, 3, 3, 3, 3, 3};
System.arraycopy(uno, 0, dos, 0, uno.length);
for (int i=0; i <= 8; i++){
    System.out.print( dos[i] + " " );
} //Sale 1 1 2 3 3 3 3 3 3
```

Clase String

Ya sabemos declarar variables string a partir de constantes literales o del operador concatenar (+) y que crean objetos inmutables de la clase `String`:

```
String texto1 = "¡Prueba de texto!";
String texto2;
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
texto2 = "Este es un texto que ocupa " +  
        "varias líneas, no obstante se puede " + "encadenar";
```

Otras formas de crearlos es usando alguno de sus constructores:

```
char[] palabras = {'P','a','l','a','b','r','a'}; // array de char  
String cadena = new String(palabras);  
byte[] datos = {97, 98, 99}; // array de bytes  
// Si no indicas el charset, se utiliza ASCII  
String codificada = new String (datos, "8859_1");
```

Lista completa de métodos (ya comentamos algunos, los repetimos, es algo que se utiliza mucho en los programas):

MÉTODO	DESCRIPCIÓN
<code>char charAt(int i)</code>	carácter que está en la posición i
<code>int compareTo(String s)</code>	Compara las dos cadenas. Devuelve un valor menor que cero si la cadena s es mayor que la original, 0 si son iguales y devuelve un valor mayor que cero si s es menor que la original
<code>int compareToIgnoreCase(String s)</code>	Compara dos cadenas, pero no tiene en cuenta si el texto es mayúsculas o no.
<code>String concat(String s)</code>	Añade la cadena s a la cadena original.
<code>String copyValueOf(char[] data)</code>	Produce un String igual al array de caracteres data.
<code>boolean endsWith(s)</code>	true si la cadena termina con el texto s.
<code>boolean equals(s)</code>	Compara ambas cadenas, true si son iguales.
<code>boolean equalsIgnoreCase(s)</code>	Compara ambas cadenas sin tener en cuenta mayúsculas y minúsculas.
<code>byte[] getBytes()</code>	Devuelve array de bytes de los códigos de cada carácter del String



UNIDAD 4. Programación Modular, Clases y Objetos.

MÉTODO	DESCRIPCIÓN
<code>void getBytes(int si, int sf, char[] d, int di);</code>	Almacena contenido de la cadena en el array de caracteres d. Toma los caracteres desde la posición si hasta la posición sf y los copia en el array desde la posición di
<code>int indexOf(s)</code>	Devuelve la posición en la cadena del texto s.
<code>int indexOf(s, pos)</code>	Devuelve la posición en la cadena del texto s, empezando a buscar desde la posición pos
<code>int lastIndexOf(s)</code>	Devuelve la última posición en la cadena del texto s
<code>int lastIndexOf(s, pos)</code>	última posición en la cadena del texto s, empezando a buscar desde la posición pos
<code>int length()</code>	longitud de la cadena
<code>boolean matches(er)</code>	verdadero si el String cumple la expresión regular
<code>String replace(ca, cn)</code>	cadena idéntica al original pero que ha cambiando los caracteres iguales a ca por cn
<code>String replaceFirst(s1, s2)</code>	Cambia la primera aparición de la cadena s1 por la cadena s2
<code>String replaceAll(s1, s2)</code>	Cambia la todas las apariciones de la cadena s1 por la cadena s2
<code>String startsWith(s)</code>	true si la cadena comienza con el texto s.
<code>String substring(p1, p2)</code>	texto que va desde posiciones p1 a p2-1.
<code>char[] toCharArray()</code>	array de caracteres a partir de la cadena dada
<code>String toLowerCase()</code>	Convierte la cadena a minúsculas



UNIDAD 4. Programación Modular, Clases y Objetos.

MÉTODO	DESCRIPCIÓN
<code>String toLowerCase(Locale loc)</code>	Lo mismo pero siguiendo las instrucciones del argumento local
<code>String toUpperCase()</code>	Convierte la cadena a mayúsculas
<code>String toUpperCase(Locale loc)</code>	Lo mismo pero siguiendo las instrucciones del argumento local
<code>String trim()</code>	Elimina los blancos que tenga la cadena tanto por delante como por detrás
<code>static String valueOf(e)</code>	Devuelve la cadena que representa el valor e. Si e es booleano, por ejemplo, devolvería una cadena con el valor true o false.

StringBuilder

Los `String` son objetos inmutables (una vez creados no pueden cambiar). Un `string` se puede reconstruir a partir del operador `+`, pero no es eficiente porque implica crear un nuevo `string` que es una copia del `string` original (esto tarda demasiado tiempo) con el nuevo carácter añadido:

```
str = str + caracter; // Copiar str y crear un nuevo objeto
```

Si tu programa necesita modificar `strings` añadiendo partes, tendrá que crear y destruir muchos objetos para conseguirlo y necesita mucha cantidad de procesamiento (ineficiente). Para solucionarlo está la clase **StringBuilder** del paquete `java.lang`. Ejemplo:

```
import java.lang.StringBuilder;
StringBuilder builder = new StringBuilder();
```

Igual que un `String`, un `StringBuilder` contiene una secuencia de caracteres. Sin embargo es posible añadirle nuevos trozos al final sin necesidad de copiar la cadena de letras anterior en nuevo objeto. Es



UNIDAD 4. Programación Modular, Clases y Objetos.

decir son versiones mutables de los String. Si *x* es cualquier valor de cualquier tipo y *builder* es una referencia a un objeto *StringBuilder*, el comando **`builder.append(x)`** añade la representación en string de *x* al final de la cadena de *builder* de forma eficiente.

Puede utilizarse muchas veces hasta acabar de construir el *StringBuilder*. Al final, se convierte *builder* en un String con la función **`builder.toString()`**. Un *StringBuilder* tiene los mismos métodos que los Strings más algunos que permiten realizar modificaciones a la secuencia de caracteres. Estos nuevos métodos son:

MÉTODO	DESCRIPCIÓN
<code>insert(p, c)</code>	Inserta cadena/carácter en la posición indicada. Ej: <pre>StringBuilder sb = new StringBuilder("Hola"); sb.insert(0, ' '); // " Hola"</pre>
<code>delete(i, f)</code> <code>deleteCharAt(p)</code>	Elimina caracteres que hay entre dos posiciones o en una posición. Ej: <pre>StringBuilder sb = new StringBuilder("Hola caracola"); sb.delete(6, 8); // "Hola ccola"</pre>
<code>replace(i,f,c)</code>	Reemplazar los caracteres que hay entre dos posiciones por otros diferentes. Ej: <pre>StringBuilder sb = new StringBuilder("Hola"); sb.replace(1,3, "elad"); // "Helada"</pre>
<code>setCharAt(p,c)</code>	Cambia el carácter de la posición <i>a</i> por <i>c</i>
<code>charAt()</code> , <code>indexOf()</code> , <code>length()</code> ...	Muchos métodos iguales que los de la clase String

Random

Permite generar números pseudoaleatorios y está en **`java.util.Random`**. Ya hemos usado en algunos ejemplos la función `Math.random()` pero en



UNIDAD 4. Programación Modular, Clases y Objetos.

algunas aplicaciones los valores que genera `Math.random()` no son suficientemente aleatorios.

Un objeto de tipo `Random` puede generar enteros además de números reales. Creamos un objeto generador de números `gn` de esta forma:

```
Random gn = new Random();
```

si `N` es un número entero positivo, la llamada `gn.nextInt(N)` devuelve un entero aleatorio entre 0 y `N-1`. Por ejemplo:

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

int dado1 = (int)(6 * Math.random()) + 1;
int dado2 = gn.nextInt(6) + 1; // mejores propiedades aleatorias
```

También tienes otros métodos para generar por ejemplo números aleatorios que sigan una distribución gaussiana en vez de uniforme, etc.

Color

Está en el paquete `java.awt`, y representa un color con el que se puede dibujar. Tiene constantes como `Color.RED`. También es posible crear nuevos colores además de los que tiene predefinidos. Tiene muchos constructores. Uno de ellos es `new Color(r,g,b)`, que acepta 3 enteros entre 0 y 255 y representa la cantidad de rojo, verde y azul del color.

Otro constructor es `new Color(r,g,b,t)`, que añade un cuarto parámetro, otro número entre 0 y 255 que indica la transparencia (a mayor valor más opacidad, a menor valor más transparencia).

Otro constructor tiene un solo entero en el que puedes pasar los componentes con 2 cifras hexadecimales cada una, `new Color(0x8B5413)` crea un color marrón

Un objeto `Color` solo tiene unos cuantos métodos getters: `getRed()`,



UNIDAD 4. Programación Modular, Clases y Objetos.

etc. y no tiene setters (es inmutable, como los String).

Arrays

Arrays es una clase estática de `java.util` creada para ofrecer algunos métodos que trabajan sobre Arrays: **Arrays.método()**:

MÉTODO	DESCRIPCIÓN
fill	<p>Permite rellenar todo o parte de un array unidimensional con un determinado valor.</p> <pre>int a[] = new int[23]; Arrays.fill(a, -1); // Todo el array vale -1 Arrays.fill(a, 5, 8, -1); // De elemento 5 al 7 a -1</pre>
equals	<p>Compara dos arrays y devuelve true si son iguales. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores. A diferencia del operador de igualdad (==), este sí compara el contenido. Ej:</p> <pre>int a[] = {2, 3, 4, 5, 6}; int b[] = {2, 3, 4, 5, 6}; int c[] = a; System.out.println(a == b); // false System.out.println(Arrays.equals(a,b)); // true System.out.println(a == c); // true System.out.println(Arrays.equals(a,c)); // true</pre>
sort	<p>Ordena un array en orden ascendente. Se pueden ordenar todo el array o bien desde un elemento a otro:</p> <pre>int x[] = {4,5,2,7,3,8,2,3,9,5}; Arrays.sort(x, 2, 5); // queda {4 5 2 3 7 8 2 3 9 5} Arrays.sort(x); // Estará completamente ordenado</pre>
binarySearch	<p>Permite buscar un elemento de forma ultrarrápida en un array ordenado (en un array desordenado sus resultados son impredecibles). Devuelve el índice en el que está el elemento. Ej:</p>



UNIDAD 4. Programación Modular, Clases y Objetos.

MÉTODO	DESCRIPCIÓN
	<pre>int x[] = {1,2,3,4,5,6,7,8,9,10,11,12}; Arrays.sort(x); System.out.println(Arrays.binarySearch(x,8)); // 7</pre>
copyOf	<p>Crea una copia de un array. Usa dos parámetros, el array a copiar y el tamaño que tendrá el array resultante. Si el tamaño es menor que el array original, sólo obtiene copia de los primeros elementos (tantos como indique el tamaño) y si el tamaño es mayor que el original, devuelve un array en el que los elementos que superan al original se rellenan con el valor por defecto (dependiendo del tipo de datos del array).</p> <pre>int a[] = {1,2,3,4,5,6,7,8,9}; int b[] = Arrays.copyOf(a, a.length); int c[] = Arrays.copyOf(a, 12); int d[] = Arrays.copyOf(a, 3); //d es {1,2,3}</pre>
copyOfRange	<p>Como la anterior, sólo que indica con dos números de qué a qué elemento se hace la copia:</p> <pre>int a[]= {1,2,3,4,5,6,7,8,9}; int b[]= Arrays.copyOfRange(a, 3, 6); //b es {4,5,6}</pre>

EXPRESIONES REGULARES (Pattern, Matcher)

Java ofrece las clases **Pattern** y **Matcher** contenidas en el paquete `java.util.regex.*`. La clase **Pattern** se utiliza para procesar la expresión regular y "compilarla" (verificar que es correcta y dejarla lista para su utilización). La clase **Matcher** sirve para comprobar si una cadena cualquiera sigue o no un patrón. Veamoslo con un ejemplo:

```
Pattern p = Pattern.compile("[01]+"); // Valida y procesa patrón
Matcher m = p.matcher("00001010");
if ( m.matches() )
    System.out.println("Si, contiene el patrón");
else
    System.out.println("No, no contiene el patrón");
```



UNIDAD 4. Programación Modular, Clases y Objetos.

En el ejemplo, el método estático **compile()** de la clase **Pattern** permite crear un patrón, dicho método compila la expresión regular pasada por y genera una instancia de **Pattern** (p en el ejemplo). El patrón p se puede usar varias veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método **matcher()**, que combina el patrón con la cadena de entrada y genera una instancia de la clase **Matcher** (m en el ejemplo).

La clase **Matcher** contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- **m.matches()**. true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.
- **m.lookingAt()**. true si el patrón se ha encontrado al principio de la cadena. A diferencia del método **matches()**, la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.
- **m.find()**. true si el patrón existe en algún lugar de la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y false en caso contrario, pudiendo tener más de una coincidencia. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos **m.start()** y **m.end()**, para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método **find()** irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método **find()**, para que vuelva a comenzar por la primera coincidencia, invocando el método **m.reset()**.

Una expresión regular es una expresión textual que utiliza símbolos especiales para hacer búsquedas avanzadas. Las expresiones regulares



UNIDAD 4. Programación Modular, Clases y Objetos.

pueden contener:

- **Caracteres.** Como a, s, ñ, ... y se los interpreta tal cual. Si una expresión regular contiene sólo un carácter, matches devolvería verdadero si el texto contiene sólo ese carácter. Si se ponen varios, obliga a que el texto tenga exactamente esos caracteres.
- **Principio y final de línea de entrada:** el acento circunflejo (^) al principio de la expresión regular indica el principio y el dólar (\$) el final de la entrada.
- **Caracteres de control:** \\d (dígito, equivale a [0-9]), \\D (cualquier cosa menos un dígito, equivale a [^0-9]), \\s (espacio en blanco, tabulador, salto), \\S (cualquier cosa menos un espacio, \\n, ...)
- **Caracteres opcionales.** Se ponen entre corchetes. Por ejemplo [abc] significa a, b ó c. Lo que vaya entre corchetes al final representa un solo carácter.
- **Negación de caracteres.** Funciona al revés, no pueden estar los caracteres indicados. Se pone con corchetes dentro de los cuales se pone el carácter circunflejo (^) al principio. [^abc] significa ni a ni b ni c.
- **Rangos.** Se ponen con guiones entre corchetes. Por ejemplo [a-z] significa: cualquier carácter de la 'a' hasta la 'z'.
- **Intersección.** Usa &&. Por ejemplo [a-x&&r-z] significa de la r a la x (intersección de ambas expresiones).
- **Substracción.** Ejemplo [a-x&&[^cde]] significa de la a hasta la x excepto la c, d ó e.
- **Cualquier carácter.** Se hace con el símbolo punto (.)
- **Opcional.** El símbolo ? sirve para indicar que la expresión que le



UNIDAD 4. Programación Modular, Clases y Objetos.

antecede puede aparecer una o ninguna veces. Por ejemplo "a?" indica que puede aparecer la letra a o no.

- **Repetición.** Se usa con el asterisco (*). Indica que la expresión puede repetirse cero o varias veces.
- **Repetición obligada.** Lo hace el signo +. La expresión se repite una o más veces (pero al menos una).
- **Repetición un número exacto de veces.** Un número entre llaves indica las veces que se repite la expresión. Por ejemplo `\d{7}` significa que el texto tiene que llevar siete números (siete cifras del 0 al 9). Con una coma significa al menos, es decir `\d{7,}` significa al menos siete veces (podría repetirse más veces). Si aparece un segundo número indica un máximo número de veces `\d{7,10}` significa de siete a diez veces.
- **Paréntesis:** permiten aplicar el operador de la derecha, no al carácter que resulte de la expresión izquierda, sino a todo lo que resulte dentro del paréntesis. Por ejemplo `"#[01]{2,3}"` representa las cadenas `#00`, `#11`, `#000` y `#111`. Si usamos paréntesis: `"(#[01]){2,3}"` representa `#0#0`, `#1#1`, `#0#0#0` y `#1#1#1`.

Ejemplos:

- `[a-z]{1,4}[0-9]+` textos que comienzan por de una a 4 letras seguidas de al menos un dígito numérico o más.
- `[XYxy]?[0-9]{1,9}[A-Za-z]` representa un DNI o un NIE.

EJEMPLO 24: Comprobar si el texto que se lee con `JOptionPane` empieza con dos guiones, le siguen tres números y luego una o más letras mayúsculas. De no ser así, el programa vuelve a pedir escribir el texto:

```
import javax.swing.JOptionPane;
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
public class ExpRegu {  
  
    public static void main(String[] args) {  
        String respuesta;  
        do {  
            respuesta= JOptionPane.showInputDialog("Escribe código: ");  
            if( respuesta.matches("--[0-9]{3}[A-Z]+") == false) {  
                JOptionPane.showMessageDialog(null, "Texto no casa");  
            }  
        } while( respuesta.matches("--[0-9]{3}[A-Z]+") == false);  
        JOptionPane.showMessageDialog(null, "Expresión correcta!");  
    }  
}
```

EJERCICIO 7: Modifica el programa del ejemplo anterior y crea la clase `TestExpReg.java` de forma que pregunte por una expresión regular usando `JOptionPane.showInputDialog` antes de entrar en el bucle. Y luego, no deje de preguntarr por texto para comprobar si el texto leído casa con la expresión o no (debe verse la expresión regular usada). Indica si casa con `JOptionPane.showMessageDialog`. Sale del bucle cuando el texto leído sea vacío.

BigDecimal

Escribe el siguiente código:

```
double centimo = 0.01;  
double suma = centimo + centimo + centimo + centimo + centimo +  
                centimo;  
System.out.println( suma );
```

¿Que imprime? Si has contestado 0.06 puede que te equivoques. Se imprime: 0.060000000000000005. Ahora escribe este:

```
java.math.BigDecimal centimo = new java.math.BigDecimal("0.01");  
java.math.BigDecimal suma = centimo.add(centimo).add(centimo)  
                                .add(centimo).add(centimo).add(centimo);  
System.out.println(suma);
```



UNIDAD 4. Programación Modular, Clases y Objetos.

¿Que imprime?

Las computadoras no cuentan como nosotros, "cuentan en binario" y nosotros contamos en decimal. Cuando usamos una variable float o double, estamos dejándole al microprocesador el trabajo de efectuar los cálculos directamente (con su coprocesador matemático) pero como las computadoras "piensan" en binario, **cometen errores de precisión diferentes a los nuestros**.

Por ejemplo, nosotros los humanos, tenemos un problema para representar valores como la tercera parte de "1" ($1/3$) y la escribimos: 0.3333333333333333... (hasta el infinito), en nuestro sistema decimal (base 10) no hay manera de representar un tercio. Sin embargo, si nuestro sistema de numeración fuera base "9", entonces representaríamos un un tercio con "0.3" (y sería más preciso que cualquier lista larga de números "3" en base 10).

En binario (base 2), no se puede representar $1/10$ (la décima parte de 1) ni $1/100$ (la centésima parte de 1) y por eso si escribimos el valor "0.01" la CPU lo expresa como 110011001100110011001100110011... (hasta el infinito) pero como no tiene infinita memoria, efectuó el cálculo incorrectamente (desde nuestro punto de vista "decimal").

Si haces programas que manipulen dinero, estos pequeños errores de representación se van acumulando en los cálculos y acaban por generar resultados que no coinciden con las operaciones que normalmente realizamos los humanos. Para resolver este problema, en Java se incluye la **clase BigDecimal**, que a cambio de realizar los cálculos más lentos que usando "Double/double" (los cálculos los hace el hardware) los realiza del mismo modo que los humanos (en base 10 y por software) y así evitamos el problema de los errores por representación en base 2. Algunas CPU también implementan



UNIDAD 4. Programación Modular, Clases y Objetos.

operaciones base 10 por hardware. Vamos a ver un código de ejemplo que muestra como usar la clase (los objetos `BigDecimal` son inmutables, una vez creados no se pueden modificar, aunque sí generar nuevos a partir de los que hay).

EJEMPLO 25: usar la clase `BigDecimal`.

```
// Ejemplo de programa Java usando BigDecimal
import java.math.BigDecimal;

public class BigDecimalEjemplo {
    public static void main(String[] args) {
        // Crea dos valores BigDecimal
        BigDecimal bd1 = new BigDecimal("124567890.0987654321");
        BigDecimal bd2 = new BigDecimal("987654321.123456789");
        // Sumarlos
        bd1 = bd1.add(bd2);
        System.out.println("BigDecimal1 = " + bd1);
        // Multiplicar
        bd1 = bd1.multiply(bd2);
        System.out.println("BigDecimal1 = " + bd1);
        // Restar
        bd1 = bd1.subtract(bd2);
        System.out.println("BigDecimal1 = " + bd1);
        // Dividir
        bd1 = bd1.divide(bd2);
        System.out.println("BigDecimal1 = " + bd1);
        // Potencia de 2
        bd1 = bd1.pow(2);
        System.out.println("BigDecimal1 = " + bd1);
        // Cambiar el signo
        bd1 = bd1.negate();
        System.out.println("BigDecimal1 = " + bd1);
    }
}
```

Algunas operaciones de `BigDecimal` comparándolas con `double`:

- Declaración de variables:
`double a, b;`
`BigDecimal A, B;`
- Inicialización:



UNIDAD 4. Programación Modular, Clases y Objetos.

```
a = 5.4;
b = 2.3;
A = BigDecimal.valueOf(5.4);
B = BigDecimal.valueOf(2.3);
// También a partir de String:
A = new BigDecimal("5.4");
B = new BigDecimal("1238126387123.1234");

//Hay constantes predefinidas:
A = BigDecimal.ONE; // También ZERO y TEN
```

- **Operaciones:** `add()`, `subtract()`, `multiply()`, `divide()`, `pow()`, ...

Mira la documentación de Java para una lista más completa de operaciones.

4.8.3 CLASES INMUTABLES Y SINGLETON.

CLASES INMUTABLES

Bien por razones técnicas o bien por que necesitamos mantener la integridad de los objetos en ciertos escenarios, habrá situaciones donde los objetos de una clase no se puedan modificar.

Por ejemplo las aplicaciones multi-proceso pueden ser un dolor de cabeza para los desarrolladores debido a que es necesario proteger el estado de los objetos de modificaciones simultáneas por parte de varios hilos concurrentes. Para conseguirlo se utilizan **bloques de Sincronización** en aquellos lugares donde el proceso modifica el estado. Este mecanismo tiene varios inconvenientes:

- **Propenso a cometer errores:** te despistas y no bloqueas en un lugar el acceso simultáneo y quizás el error solo aparece de semana en semana, con lo cual es difícil de detectar.
- **Penaliza el rendimiento:** el proceso que necesite acceder a un objeto que ya está siendo utilizado queda bloqueado hasta que el objeto se libere.



UNIDAD 4. Programación Modular, Clases y Objetos.

Con las clases inmutables, los estados nunca se modifican. Cada modificación del estado acaba en una nueva instancia, por lo tanto, cada hilo utiliza una instancia diferente y los desarrolladores no deben preocuparse por las modificaciones de procesos concurrentes.

Las clases inmutables deben:

- Tener sus atributos privados (por su visibilidad no pueden ser accedidos desde el exterior)
- No tener métodos modificadores (setters) por tanto una vez establecido el estado del objeto a través de un constructor sus atributos no cambiarán.

EJEMPLO 26: Implementación mejorable de una clase inmutable:

```
public class AlumnoInmutable {  
    private int id;           // Tipo primitivo  
    private String nombre;    // Objeto inmutable  
    private int[] notas;      // Objeto mutable  
  
    public AlumnoInmutable(String nombre, int id, int[] notas) {  
        this.nombre = nombre;  
        this.id = id;  
        this.notas = notas;    // peligro!! entra del exterior  
    }  
  
    public getNombre(){ return nombre; }  
    public getId(){ return id; }  
    public getNotas(){return notas;} //peligro!! sale al exterior  
}
```

Sin embargo, realmente, lograr que una clase sea inmutable no es tan sencillo. Además de lo anterior, debemos conseguir:

- Bloquear la herencia (si dejas hacer clases hijas, estas pueden ser gamberras y no respetar la inmutabilidad).
- Los atributos deben ser finales (para cambiarlos una sola vez).
- Si el estado del objeto tiene otros objetos mutables, no



UNIDAD 4. Programación Modular, Clases y Objetos.

aceptar objetos externos en constructores (si compartes con el exterior, desde fuera pueden modificar datos) ni devolver objetos mutables en los getters (si los pasas, los compartes con el exterior y los podrán cambiar). La solución es hacer copias de estos objetos cuando se intercambian ("entran al" o "salen del" objeto inmutable).

EJEMPLO 27: Implementación mejorada de una clase inmutable:

```
public final class AlumnoInmutable {  
    private final int id;           // Tipo primitivo  
    private final String nombre;    // Objeto inmutable  
    private final int[] notas;      // Objeto mutable  
  
    public Alumno(String nombre, int id, int[] notas) {  
        this.nombre = nombre;  
        this.id = id;  
        // Si el array fuese de objetos habría que clonarlos tb!!  
        this.notas = notas.clone(); // Se queda con una copia  
    }  
  
    public getNombre() { return nombre; }  
    public getId() { return id; }  
    public getNotas() { return notas.clone(); } // devuelve copia  
}
```

CLASES SINGLETON

Una clase de tipo singleton es aquella que se utiliza cuando interesa que **solo pueda tener una instancia** (que solo se pueda crear un único objeto suyo) en cada aplicación. Igual que el caso de las clases inmutables, no es que en el lenguaje haya palabras clave que te permitan definirla, sino que utilizando los elementos del lenguaje, consigues este comportamiento. A esto se le llama patrón de diseño, una forma de hacer las cosas.

La clase ofrece un único punto de entrada para crear objetos y esto le permite controlar el número de los que se crean. Lo consiguen haciendo



UNIDAD 4. Programación Modular, Clases y Objetos.

privado el constructor y ofreciendo al exterior un método estático que crea la instancia la primera vez o devuelve una referencia las siguientes veces.

EJEMPLO 28: Implementación mejorable de una clase singleton:

```
public class EjemploSingleton {  
    // El constructor privado: no se usa desde fuera.  
    private EjemploSingleton() {  
        System.out.println("Nace una instancia.");  
    }  
  
    // Una referencia a la única instancia  
    private static EjemploSingleton hijoUnico;  
  
    // Un método de la clase que da acceso al hijo  
    public static EjemploSingleton getInstance() {  
        if(hijoUnico == null) hijoUnico = new EjemploSingleton();  
        return hijoUnico;  
    }  
  
    // Otros elementos: datos, métodos...  
}
```

La implementación anterior puede fallar si la ejecutamos en una aplicación con varios threads, porque llamar a **getInstance()** desde varios threads al mismo tiempo puede acabar creando varias instancias.

Una solución, puede ser usar la palabra clave **synchronized** en el método **getInstance()**, de esta forma solo un thread puede entrar en el código del método al mismo tiempo (es como el cerrojo del servicio de un local), el resto de los que quieran entrar quedan encolados (bloqueados a la espera de poder entrar cuando el cerrojo se abra) evitando que varios ejecuten su código a la vez.

EJEMPLO 29: Implementación mejorada de una clase singleton:



UNIDAD 4. Programación Modular, Clases y Objetos.

```
public class EjemploSingleton {  
    // El constructor privado: no se usa desde fuera.  
    private EjemploSingleton() {  
        System.out.println("Nace una instancia.");  
    }  
  
    // Una referencia a la única instancia  
    private static EjemploSingleton hijoUnico;  
  
    // Un método de la clase que da acceso al hijo  
    public static synchronized EjemploSingleton getInstancia() {  
        if(hijoUnico == null) hijoUnico = new EjemploSingleton();  
        return hijoUnico;  
    }  
  
    // Otros elementos: datos, métodos...  
}
```

Otra posible solución es usar una clase interna (se instancia una sola vez al referenciarla por primera vez) a la que llamaremos creadora y aprovechando este comportamiento, asignaremos la referencia al hijo único en ese momento, así aunque varios threads entren al mismo tiempo en `getInstancia()` solo uno de ellos provocará la creación del hijo.

EJEMPLO 30: Implementación mejorada de clase singleton sin usar bloqueos:

```
public class EjemploSingleton {  
    // La clase interna estática  
    public static class Creadora {  
        public static final EjemploSingleton hijoUnico =  
            new EjemploSingleton();  
    }  
  
    // El constructor privado: no se usa desde fuera.  
    private EjemploSingleton() {  
        System.out.println("Nace una instancia.");  
    }  
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
// Un método de la clase que da acceso al hijo
public static EjemploSingleton getInstancia() {
    return Creadora.hijoUnico;
}

// Otros elementos: datos, métodos...
```

4.9. MÓDULOS

En JDK 9 se añade una nueva característica a Java llamada módulos. Los módulos aportan una forma de describir las relaciones y dependencias del código que forma una aplicación. También permiten controlar qué partes son accesibles desde otros módulos y cuáles no. Usándolos se pueden crear programas más confiables y escalables.

Como regla general, los módulos son más útiles para aplicaciones de gran tamaño. Sin embargo, los pequeños programas también se benefician porque la API de Java se ha organizado en módulos. Por tanto, es posible especificar qué partes de la API necesita tu programa y cuáles no. Esto permite implementar programas con un espacio de tiempo de ejecución más pequeño, algo especialmente importante cuando se crean códigos para dispositivos pequeños.

La compatibilidad con los módulos se proporciona mediante elementos del lenguaje, incluidas **nuevas palabras clave** y mediante **mejoras en javac, java y otras herramientas**. Además, se han introducido nuevas herramientas y formatos de archivo. En resumen, los módulos constituyen una importante mejora del lenguaje Java.

QUÉ ES UN MÓDULO

Es una **agrupación de paquetes y recursos a los que se puede hacer**



UNIDAD 4. Programación Modular, Clases y Objetos.

referencia colectivamente por el nombre del módulo. Una declaración de módulo especifica el nombre de un módulo y define la relación que el módulo y sus paquetes tienen con otros módulos. Las declaraciones de módulo son declaraciones de programa en un archivo fuente de Java y son compatibles con varias palabras clave nuevas relacionadas con el módulo.

open	module	requires	transitive
exports	opens	to	uses
provides	with		

Tabla: Palabras claves reservadas (para Módulos) en Java.

Estas palabras clave se reconocen solo en el contexto de una declaración de módulo. Fuera del fichero fuente del módulo se interpretan como identificadores corrientes. Por lo tanto, la palabra `module` podría, por ejemplo, también ser utilizada como un nombre de parámetro.

Una declaración de módulo está contenida en un archivo llamado **module-info.java**. Por lo tanto, un módulo se define en un archivo fuente Java. Este archivo es compilado por `javac` en un archivo de clase y se conoce como un **descriptor de módulo** (module descriptor). El archivo `module-info.java` debe contener solo una definición de módulo. No es un archivo de propósito general.

Una declaración de módulo comienza con la palabra clave **module**. Aquí está su sintaxis general:

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



UNIDAD 4. Programación Modular, Clases y Objetos.

El nombre del módulo debe ser un identificador de Java válido o una secuencia de identificadores separados por puntos. La definición del módulo se especifica dentro de las llaves. Aunque la definición puede estar vacía (lo que crea una declaración que simplemente nombra el módulo), típicamente especifica una o más cláusulas que definen las características del módulo.

Antes de continuar, unas palabras sobre los nombres de los módulos. En el ejemplo siguiente, el nombre del módulo `appfuncs` es el prefijo del nombre de un paquete que contiene (`appfuncs.funcsimples`). Esto no es obligatorio, pero ayuda a indicar claramente a qué módulo pertenece un paquete. En general, los nombres cortos y simples son útiles.

La forma sugerida es usar el nombre de dominio inverso: el nombre de dominio inverso del dominio que "posee" el proyecto se usa como un prefijo para el módulo. Por ejemplo, un proyecto asociado con la empresa `damlabs.com` usaría `com.damlabs` como el prefijo del módulo. Lo mismo ocurre con los nombres de los paquetes. Pero como los módulos son una novedad en Java, las convenciones de nombres pueden evolucionar con el tiempo.

Entre las capacidades de un módulo hay dos que son clave:

- La primera es la de especificar que requiere otro módulo. En otras palabras, un módulo puede indicar que depende de otro. Una relación de dependencia se especifica mediante el uso de una instrucción **require**. La presencia del módulo requerido se comprueba en tiempo de compilación y de ejecución.
- La segunda es la de controlar cuáles de sus paquetes (si tiene alguno), son accesibles por otro módulo. Esto se logra mediante el uso de la palabra clave **exports**. Los tipos **public** y **protected** dentro de un paquete solo son accesibles desde otros módulos si



UNIDAD 4. Programación Modular, Clases y Objetos.

se exportan explícitamente.

EJEMPLO 31. Un módulo básico.

El siguiente ejemplo muestra estas dos características al crear una aplicación modular que demuestra algunas funciones matemáticas simples. Aunque esta aplicación es muy pequeña, ilustra los conceptos básicos y los procedimientos necesarios para crear, compilar y ejecutar código basado en módulos y sin embargo el enfoque general también se aplica a las aplicaciones más grandes del mundo real. De hecho, en aplicaciones grandes el IDE que se utilice se encargará de ejecutar los comandos apropiados al crear los elementos como módulos, paquetes, etc. en vez de hacer las carpetas a mano. La aplicación define dos módulos:

- El primer módulo se llama **appinicio**. Contiene un paquete llamado appinicio.midemoappmod que define el punto de entrada de la aplicación en una clase llamada MiAppModDemo. Por lo tanto, MiAppModDemo contiene el método `main()` de la aplicación.
- El segundo módulo se llama **appfuncs**. Contiene un paquete llamado appfuncs.funcsimples que incluye la clase FuncsMateSimples. Esta clase define 3 métodos estáticos que implementan algunas funciones matemáticas simples. Toda la aplicación estará contenida en un árbol de directorios que comienza en `miappmodulo`.

PASO 1. Crear la estructura de la aplicación

Comenzamos creando los directorios de códigos fuente necesarios siguiendo estos pasos:

1. Crea un directorio llamado `miappmodulo`. Este es el directorio de nivel superior para toda la aplicación.
2. En `miappmodulo`, crea un subdirectorio llamado `appsrc`. Este es



UNIDAD 4. Programación Modular, Clases y Objetos.

el directorio de nivel superior para el código fuente de la aplicación.

3. En appsrc, crea un subdirectorio appinicio. En este directorio, crea un subdirectorio también llamado appinicio. Bajo este directorio, crea el directorio midemoappmod. Por lo tanto, comenzando con appsrc, habrá creado este árbol:

```
appsrc\appinicio\appinicio\midemoappmod
```

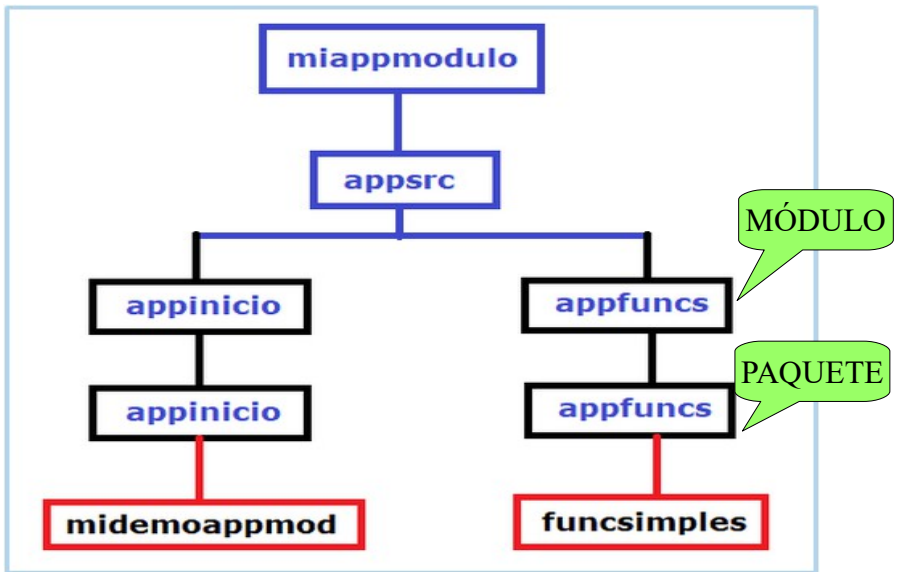
4. También en appsrc, cree el subdirectorio appfuncs. Bajo este directorio, cree un subdirectorio llamado appfuncs. Bajo este directorio, crea el directorio llamado funcsimples. Por lo tanto, comenzando con appsrc, habrá creado este árbol:

```
appsrc\appfuncs\appfuncs\funcsimples
```

Después de configurar estos directorios creamos los ficheros fuente. Usaremos 4 archivos fuente. Dos son los archivos fuente que definen la aplicación y los otros dos la especificación de ambos módulos. Su árbol de directorios debería verse como el que se muestra aquí:



UNIDAD 4. Programación Modular, Clases y Objetos.



Primer archivo: El primero es [FuncsMateSimples.java](#), que se muestra a continuación. Observe que [FuncsMateSimples.java](#) está empaquetado en [appfuncs.funcsimples](#).

```
package appfuncs.funcsimples;

public class FuncsMateSimples{

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

    // Devuelve el divisor positivo más pequeño común de a y b
    public static int divPeq(int a, int b){
        a = Math.abs(a);
        b = Math.abs(b);
        int min = a < b ? a : b;
        for(int i = 2; i <= min/2; i++){
            if( esDivisor(i,a) && esDivisor(i,b) )
                return i;
        }
    }
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```

    }
    return 1;
}

// Devuelve el mayor factor positivo común de a y b
public static int divGra(int a, int b){
    a = Math.abs(a);
    b = Math.abs(b);
    int min = a < b ? a : b;
    for(int i = min/2; i>=2; i--){
        if( esDivisor(i,a) && esDivisor(i,b) )
            return i;
    }
    return 1;
}
}

```

FuncsMateSimples define 3 funciones matemáticas estáticas (static) simples. El primero, esDivisor(), divPeq() y divGra(). Este archivo debe colocarse en el siguiente directorio:

appsrc\appfuncs\appfuncs\funcsimples

Este es el directorio del package `appfuncs.funcsimples`

Segundo archivo. es MiAppModDemo.java, que se muestra a continuación. Utiliza los métodos en FuncsMateSimples. Observe que está empaquetado en appinicio\midemoappmod. También tenga en cuenta que importa la clase FuncsMateSimples porque depende de FuncsMateSimples para su funcionamiento.

```

// Demo de una aplicación simple basada en módulos.
package appinicio.midemoappmod;

import appfuncs.funcsimples.FuncsMateSimples;

public class MiAppModDemo {
    public static void main(String[] args) {
        if( FuncsMateSimples.esDivisor(2,10) );
            System.out.println("2 es divisor de 10");
        System.out.println("Menor divisor positivo de 33 y 99: "

```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
        + FuncsMateSimples.divPeg(33,99) );  
System.out.println("Mayor divisor positivo de 33 y 99: "  
        + FuncsMateSimples.divGra(33,99) );  
    }  
}
```

Este archivo debe colocarse en el siguiente directorio:

appsrc\appinicio\appinicio\midemoappmod

Este es el directorio del package `appinicio.midemoappmod`;

Archivos `module-info.java`. A continuación, debe agregar archivos `module-info.java` para cada módulo. Estos archivos contienen las definiciones de los módulos. Primero el que define el módulo appfuncs:

```
// Definición del módulo para el módulo de funciones.  
module appfuncs{  
    // Exporta los paquetes appfuncs.funcsimples.  
    exports appfuncs.funcsimples;  
}
```

Tenga en cuenta que `appfuncs` exporta el paquete `appfuncs.funcsimples`, que lo hace accesible a otros módulos. Este archivo debe colocarse en este directorio:

appsrc\appfuncs

Por lo tanto, va en el directorio del módulo `appfuncs`, que está encima de los directorios del paquete.

Por último se añade el archivo `module-info.java` del módulo **appinicio**. Tenga en cuenta que appinicio requiere el módulo appfuncs.

```
// Definición del módulo de aplicación principal (main).  
module appinicio {  
    // Requiere el módulo appfuncs.  
    requires appfuncs;  
}
```

Este archivo debe colocarse en su directorio de módulos:



UNIDAD 4. Programación Modular, Clases y Objetos.

appsrc\appinicio

EJEMPLO 32. Compilar y ejecutar el primer ejemplo de módulos.

Antes de examinar las declaraciones: `require`, `exports` y `module` más de cerca, primero compilemos y ejecutemos el ejemplo anterior. Asegúrese de haber creado correctamente los directorios y dejado cada archivo en su directorio apropiado, como se acaba de explicar.

Como todos los demás programas de Java, los programas basados en módulos se compilan utilizando `javac`. El proceso es fácil, con la diferencia principal de que generalmente se especificará explícitamente una ruta de módulo. Una ruta de módulo le dice al compilador dónde se ubicarán los archivos compilados. Ejecute los comandos `javac` desde el directorio `miappmodulo` para que las rutas sean correctas. Recuerde que `miappmodulo` es el directorio de nivel superior para toda la aplicación basada en módulos.

Para comenzar, compile el archivo `FuncsMateSimples.java`, usando el comando:

```
javac -d appmodules/appfuncs  
      appsrc\appfuncs\appfuncs\funcsimples\FuncsMateSimples.java
```

La opción `-d` le dice a `javac` dónde colocar el archivo `.class` de salida.

Este comando creará automáticamente los directorios del paquete de salida para `appfuncs.funcsimples` bajo `appmodules\appfuncs` según sea necesario.

A continuación, aquí está el comando `javac` que compila el archivo `module-info.java` para el módulo `appfuncs`:

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



UNIDAD 4. Programación Modular, Clases y Objetos.

Esto coloca el archivo `module-info.class` en el directorio `appmodules\appfuncs`.

Aunque el proceso anterior de dos pasos funciona, es más fácil compilar el archivo `module-info.java` de un módulo y sus archivos fuente en una sola línea de comandos. Aquí, los dos comandos anteriores `javac` se combinan en uno:

```
javac -d appmodules/appfuncs appsrc\appfuncs\module-info.java
appsrc\appfuncs\appfuncs\funcsimples\FuncsMateSimples.java
```

En este caso, cada archivo compilado se coloca en su módulo o directorio de paquetes apropiado. Ahora, compile los archivos `module-info.java` y `MiAppModDemo.java` para el módulo `appinicio`, usando este comando:

```
javac --module-path appmodules -d appmodules/appinicio
appsrc\appinicio\module-info.java
appsrc\appinicio\appinicio\midemoappmod\MiAppModDemo.java
```

La opción `--module-path` especifica la ruta del módulo, que es la ruta en la que el compilador buscará los módulos definidos por el usuario requeridos por el archivo `module-info.java`. En este caso, buscará el módulo `appfuncs` porque lo necesita el módulo `appinicio`. Además, observe que especifica el directorio de salida como `appmodules/appinicio`. Esto significa que el archivo `module-info.class` estará en el directorio del módulo `appmodules/appinicio` y `MiAppModDemo.class` estará en el directorio del paquete `appmodules\appinicio\appinicio\midemoappmod`.

Una vez que haya completado la compilación, puede ejecutar la aplicación con este comando `java`:

```
java --module-path appmodules -m
appinicio/appinicio.midemoappmod.MiAppModDemo
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Aquí, la opción `--module-path` especifica la ruta a los módulos de la aplicación. Como se mencionó, `appmodules` es el directorio en la parte superior del árbol de módulos compilados. La opción `-m` especifica la clase que contiene el punto de entrada de la aplicación y, en este caso, el nombre de la clase que contiene el método `main()`. Cuando ejecutas el programa, verás el siguiente resultado:

2 es divisor de 10

El divisor positivo más pequeño entre 33 y 99 es: 3

El divisor positivo más grande entre 33 y 99 es: 11

OPCIONES REQUIRES Y EXPORT

El ejemplo anterior basado en módulos se usan las dos características fundamentales del sistema de módulos: la capacidad de especificar una dependencia y la capacidad de satisfacer esa dependencia. Estas capacidades se especifican mediante el uso de las declaraciones `requires` y `exports` dentro de una declaración `module`.

requires nombreModulo;

Aquí, `nombreModulo` especifica el nombre de un módulo requerido por el módulo en el que aparece la instrucción `require`. Esto significa que el módulo requerido debe estar presente para que el módulo actual pueda compilarse. En el lenguaje de los módulos, se dice que el módulo actual lee el módulo especificado en la instrucción `require`. En general, la declaración de requisitos le proporciona una forma de garantizar que su programa tenga acceso a los módulos que necesita.

exports nombrePaquete;

Aquí, `nombrePaquete` especifica el nombre del paquete que se exporta por el módulo en el que aparece esta declaración. Cuando un módulo exporta un paquete, hace que todos los tipos públicos y protegidos en el paquete sean accesibles para otros módulos. Además, los miembros



UNIDAD 4. Programación Modular, Clases y Objetos.

públicos y protegidos de esos tipos también son accesibles. Sin embargo, si un paquete dentro de un módulo no se exporta, entonces es privado para ese módulo, incluidos todos sus tipos públicos.

Por ejemplo, a pesar de que una clase se declara como `public` dentro de un paquete, si ese paquete no es exportado explícitamente por una declaración `exports`, entonces esa clase no es accesible para otros módulos. Es importante comprender que los tipos `public` y `protected` de un paquete, ya sean exportados o no, siempre son accesibles dentro del módulo de ese paquete. La declaración `exports` simplemente los hace accesibles a módulos externos.

Es importante recordar que las sentencias `requires` y `exports` deben aparecer sólo dentro de una declaración `module`. Además, una declaración `module` debe aparecer por sí misma en un archivo llamado `module-info.java`.

MÓDULOS `java.base` y módulos de plataforma

Los módulos de API se denominan módulos de plataforma, y todos sus nombres comienzan con el prefijo `java`. Estos son algunos ejemplos: `java.base`, `java.desktop` y `java.xml`.

El hecho de que todos los paquetes de la biblioteca Java API estén ahora en módulos da lugar a la siguiente pregunta: ¿Cómo puede el método `main()` en `MiAppModDemo` en el ejemplo anterior usar `System.out.println()` sin especificar una sentencia `requires` para el módulo que contiene la clase `System`? Obviamente, el programa no se compilará y ejecutará a menos que `System` esté presente. La misma pregunta también se aplica al uso de la clase `Math` en `FuncsMateSimples`. La respuesta a esta pregunta se encuentra en `java.base`.



UNIDAD 4. Programación Modular, Clases y Objetos.

De los módulos de plataforma, el más importante es **java.base**. Incluye y exporta aquellos paquetes fundamentales para Java, como **java.lang**, **java.io** y **java.util**, entre muchos otros. Debido a su importancia, **java.base** es automáticamente accesible para todos los módulos. Además, todos los demás módulos requieren automáticamente **java.base**.

- No es necesario incluir una declaración **java.base** en una declaración de **module**.
- De la misma manera que **java.lang** está automáticamente disponible para todos los programas sin el uso de una declaración **import**, el módulo **java.base** es automáticamente accesible para todos los programas basados en módulos sin solicitarlo explícitamente.

Debido a que **java.base** contiene el paquete **java.lang**, y **java.lang** contiene la clase **System**, MiAppModDemo en el ejemplo anterior puede usar **System.out.println()** automáticamente sin una declaración **requires** explícita. Lo mismo se aplica al uso de la clase de **Math** en FuncsMateSimples, porque la clase de **Math** también está en **java.lang**.

Por lo tanto, la inclusión automática de **java.base** simplifica la creación de código basado en módulos porque los paquetes centrales de Java son accesibles automáticamente.

EXPORTAR A UN MÓDULO ESPECÍFICO

La forma básica de la declaración **exports** hace que un paquete sea accesible para todos los demás módulos. Sin embargo, en algunas situaciones, puede ser deseable hacer que un paquete sea accesible solo para un conjunto específico de módulos, no para todos los demás módulos.



UNIDAD 4. Programación Modular, Clases y Objetos.

Por ejemplo, un desarrollador de biblioteca (Library Developer) puede querer exportar un paquete de soporte a ciertos módulos dentro de la biblioteca, pero no hacer que esté disponible para uso general. Agregar una cláusula **to** a la declaración **exports** permite lograr esto. En una declaración **exports**, la cláusula **to** especifica una lista de uno o más módulos que tienen acceso al paquete exportado. Además, solo aquellos módulos nombrados en la cláusula **to** tendrán acceso. La sintaxis de **exports** que incluye **to** es:

exports nombrePaquete to listaModulos;

listaModulos es una lista de nombres de módulos separados por comas a los que el módulo de exportación otorga acceso. Puedes probar la cláusula **to** cambiando el archivo module-info.java para el módulo appfuncs:

```
module appfuncs{
    // Exporta el paquete appfuncs.funcsimples a appinicio
    exports appfuncs.funcsimples to appinicio;
}
```

Puedes recompilar la aplicación usando este comando **javac**:

```
javac -d appmodules --module-source-path appsrc
appinicio appsrc\appinicio\appinicio\midemoappmod\MiAppModDemo.java
```

Observe que especifica la opción **-module-source-path**, esta compila automáticamente los archivos en el árbol bajo el directorio especificado, que es **appsrc** en este ejemplo. La opción **-module-source-path** se debe utilizar con la opción **-d** para garantizar que los módulos compilados se almacenen en sus directorios adecuados bajo **appmodules**.

USO DE REQUIRES TRANSITIVE



UNIDAD 4. Programación Modular, Clases y Objetos.

Considere una situación en la que hay tres módulos, A, B y C, que tienen las siguientes dependencias:

A requiere B.

B requiere C.

Dada esta situación, está claro que, dado que A depende de B y B depende de C, A tiene una dependencia indirecta de C. Siempre que A no utilice directamente ninguno de los contenidos de C, entonces simplemente puede hacer que A requiera B en su archivo de información de módulo, y haga que B exporte los paquetes requeridos por A en su archivo de información de módulo, como se muestra aquí:

```
//Archivo module-info de A:
module A{
    requires B;
}

//Archivo module-info de B:
module B{
    exports algun.paquete;
    requires C;
}
```

Aquí, *algun.paquete* es un marcador de posición para el paquete exportado por B y utilizado por A. Aunque esto funciona siempre que A no necesite usar nada definido en C, se produce un problema si A desea acceder a un tipo en C. En este caso, hay dos soluciones.

- La primera solución es simplemente agregar una declaración **requires C** al archivo de A, como se muestra aquí:

```
//El archivo module-info de A se actualizó para requerir
explícitamente C:
module A {
    requires B;
    requires C;
}
```

Esta solución ciertamente funciona, pero si B será utilizada por



UNIDAD 4. Programación Modular, Clases y Objetos.

muchos módulos, debe agregar *C* a todas las definiciones de módulos que requieren *B*. Esto no solo es tedioso; también es **propenso a errores**. Afortunadamente, hay una mejor solución.

- Puede crear una **dependencia implícita** de *C*. La dependencia implícita también se conoce como **legibilidad implícita**. Para crear una dependencia implícita, añades la palabra clave **transitive** después de **requires** en la cláusula que requiere el módulo sobre el que se necesita una legibilidad implícita. En el caso de este ejemplo, cambiarías *module-info* de *B* como aquí:

```
//Archivo module-info de B:  
module B{  
    exports algun.paquete;  
    requires transitive C;  
}
```

Aquí, *C* ahora se requiere como **transitivo**. Después de hacer este cambio, cualquier módulo que dependa de *B* también depende automáticamente de *C*. *A* tendría automáticamente acceso a *C*.

4.10 MÁS SOBRE ENUMERACIONES.

El uso de constantes simbólicas en lugar de valores literales ahorra trabajo y facilita la comprensión del código. En la unidad 2 comentamos el uso de las enumeraciones y dijimos que eran una alternativa más eficiente al uso de constantes simbólicas separadas.

Por ejemplo si queremos hacer un programa para una empresa que alquila motos y coches y donde el precio de alquiler del coche sea 100 y de la moto 50 podríamos no usar constantes simbólicas:

```
class Alquiler {
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
public static void main(String[] args)
throws IllegalArgumentException{
    System.out.print("Quiere alquilar moto o coche: ");
    String nombre = teclado.next().toUpperCase();
    int vehiculo;
    switch(nombre) {
        case "COCHE": vehiculo = 1; break;
        case "MOTO": vehiculo = 2; break;
        default: vehiculo = 0;
    }
    System.out.println("El precio de alquilar " + nombre + " es " +
        getPrecioAlquilar(vehiculo) );
}

public int getPrecioAlquilar(int vehiculo)
throws IllegalArgumentException {
    // Tendremos que validar el parámetro
    if(vehiculo < 1 || parametro > 2)
        throw new IllegalArgumentException("Vehiculo no existe");
    if(vehiculo == 1) return 100;
    if(vehiculo == 2) return 50;
}
}
```

En este ejemplo no hay mucho problema porque es pequeño y solo tenemos 2 posibilidades, pero si tenemos muchas, resulta difícil de entender y modificar. Por ejemplo, si el precio de alquilar motos cambia a 60, ¿Qué tendrías que modificar?

Otra posibilidad que mejora el código es definir constantes simbólicas aisladas para cada tipo de coche. El código similar al anterior quedaría así:

```
class Alquiler {
    // Definimos una constante simbólica para cada tipo
    public static final int COCHE = 1;
    public static final int MOTO = 2;

    public static void main(String[] args)
    throws IllegalArgumentException{
        System.out.print("Quiere alquilar moto o coche: ");
    }
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
String nombre = teclado.next().toUpperCase();
int vehiculo;
switch(nombre) {
    case "COCHE": vehiculo = COCHE; break;
    case "MOTO": vehiculo = MOTO; break;
    default: vehiculo = 0;
}
System.out.println("El precio de alquilar " + nombre + " es " +
    getPrecioAlquilar(vehiculo) );
}

public int getPrecioAlquilar(int vehiculo)
throws IllegalArgumentException {
    // Tendremos que validar el parámetro
    if(vehiculo < COCHE || parametro > MOTO)
        throw new IllegalArgumentException("Vehiculo no existe");
    if(vehiculo == COCHE) return 100;
    if(vehiculo == MOTO) return 50;
}
}
```

En esta versión el código es prácticamente el mismo con la ventaja de que hemos cambiado valores literales por constantes simbólicas, esto hace que sea más fácil de interpretar. Si te piden ahora que cambies el precio de alquilar una moto, la parte en amarillo es más sencilla de interpretar.

Las enumeraciones nos permiten usar constantes simbólicas, pero una enumeración equivale a una clase y las constantes son en realidad objetos. Como la palabra enum es casi equivalente a la palabra class y cada constante que definamos es un objeto de esa clase, por ejemplo:

```
enum Vehiculo {COCHE, MOTO}
```

Equivale internamente a:

```
/* internamente enum Vehiculo se convierte en
class Vehiculo {
    public static final Vehiculo COCHE = new Vehiculo();
    public static final Vehiculo MOTO = new Vehiculo();
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
}*/
```

Se puede declarar dentro o fuera de una clase porque ella misma es una clase. Podemos declarar variables del tipo de la enumeración que contengan algunas de sus constantes:

```
Vehiculo v = Vehiculo.COCHE;
```

Podemos imprimir una constante directamente:

```
System.out.println("Ha elegido alquilar " + v);
```

Podemos usar las variables en una sentencia switch:

```
switch(v) {  
  case COCHE: System.out.println("Buena elección");  
              break;  
  case MOTO:  System.out.println("Tenga cuidado");  
              break;  
}
```

Además podemos usar los métodos:

- `enumeracion.values()` devuelve un array con todos los valores del enum.
- `Constante.ordinal()`: índice de la constante, es decir, en qué posición aparece (0 si es la primera constante, 1 la segunda...).
- `Constante.valueOf(String s)`: devuelve la constante de la enumeración cuyo nombre coincide con el valor del String, si existe. Si no existe lanza `IllegalArgumentException`.

Y como la enumeración es en realidad una clase (de uso un poco especial, eso sí) puede tener constructores (no públicos) y métodos y variables de instancia.

Por ejemplo podemos añadir el precio de alquilar cada vehículo como un dato. En este caso debemos añadir una variable de instancia, pasar el dato al crear la constante y definir un constructor con el dato y un



UNIDAD 4. Programación Modular, Clases y Objetos.

método getter para poder consultarlo:

```
// Uso de constructor, variable de instancia y método getter
enum Vehiculo{
    COCHE(100), MOTO(10); // Al construir cada constante pasamos dato
    private int precio;    // El dato que tiene cada constante/Objeto
    private Vehiculo(int pasta){ precio = pasta; } // El constructor
    public int getPrecio(){ return precio; } // El geter
}
```

Y el código de ejemplo quedaría así con el uso de enumeraciones:

```
class Alquiler {
    enum Vehiculo{
        COCHE(100), MOTO(10); // Al construir cada constante pasamos dato
        private int precio;    // El dato que tiene cada constante
        private Vehiculo(int pasta){ precio = pasta; } // El constructor
        public int getPrecio(){ return precio; } // El geter
    }

    public static void main(String[] args)
        throws IllegalArgumentException{
        System.out.print("Quiere alquilar moto o coche: ");
        Vehiculo v = Vehiculo.valueOf( teclado.next().toUpperCase() );
        System.out.println("El precio de alquilar " + v + " es " +
            getPrecioAlquilar(v) );
    }

    public int getPrecioAlquilar(Vehiculo vehiculo) {
        // No tenemos que validar, vehiculo solo puede ser de la enum
        return vehiculo.getPrecio();
    }
}
```

Incluso podríamos suprimir el método `getPrecioAlquilar()` y usar directamente `v.getPrecio()` al imprimir.

Espero que hayas visualizado que el uso de enumeraciones a la larga hace que el código de nuestros programas sea más corto, más comprensible, más fácil de mantener y facilita la comprobación de errores.



UNIDAD 4. Programación Modular, Clases y Objetos.

4.11. EJERCICIOS.

E1. TEST

T1. Una caja negra tiene una interfaz y una implementación. Explica el significado de estos dos elementos.

T2. Una subrutina se dice que es un contrato ¿En qué consiste este contrato? (cuando quieras usarla qué debes tener presente en ese contrato). Este contrato tiene un aspecto sintáctico y otro semántico. ¿En qué consiste cada uno?

T3. Explica brevemente como las subrutinas ayudan a diseñar programas de forma top-down.

T4. ¿Para qué sirven los parámetros de una subrutina? ¿Qué diferencia hay entre los parámetros actuales y formales?

T5. ¿Qué es una API? Da un ejemplo en Java.

T6. Escribe una subrutina llamada **estrellas** que imprima una línea de asteriscos. La cantidad de estrellas será un parámetro. ¿Cómo se escribe una llamada a la subrutina desde la misma clase?

T7. Escribe una rutina **main()** que use la subrutina **estrellas** del ejercicio T6 para que muestre 10 líneas, la primera con una estrella, la segunda con dos estrellas, etc.

```
*
**
***
****
*****
*****
```



UNIDAD 4. Programación Modular, Clases y Objetos.

T8. Escribe una función llamada **cuentaLetras()** que tenga de parámetros un String y un char. Devuelve el número de veces que aparece el carácter en el String.

T9. Escribe una subrutina de 3 parámetros que sean números enteros. La subrutina debe averiguar cuál de ellos es el menor y devolverlo.

T10. Escribe una subrutina que encuentre y devuelva la media de los primeros n elementos de un array de tipo double. El array y el número n son parámetros.

T11. Explica lo que hace la siguiente función:

```
static int[] stripZeros( int[] lista ) {  
    int cont = 0;  
    for (int i = 0; i < lista.length; i++) {  
        if ( lista[i] != 0 )  
            cont++;  
    }  
    int[] nuevaLista;  
    nuevaLista = new int[cont];  
    int j = 0;  
    for (int i = 0; i < lista.length; i++) {  
        if ( lista[i] != 0 ) {  
            nuevaLista[j] = lista[i];  
            j++;  
        }  
    }  
    return nuevaLista;  
}
```

T12. Complete las siguientes oraciones:

- Un método se invoca con una _____.
- A una variable que se conoce sólo dentro del método en el



UNIDAD 4. Programación Modular, Clases y Objetos.

que está declarada, se le llama _____.

c) La instrucción _____ en un método llamado puede usarse para devolver el valor de una expresión al método que hizo la llamada.

d) La palabra clave _____ indica que un método no devuelve ningún valor.

e) Las tres formas de regresar el control de un método llamado a un solicitante son _____, _____ y _____.

f) El _____ de una declaración es la porción del programa que puede hacer referencia por su nombre a la entidad en la declaración.

g) Es posible tener varios métodos con el mismo nombre, en donde cada uno opere con distintos tipos o números de argumentos. A esta característica se le llama _____ de métodos.

T13. Escriba el encabezado para cada uno de los siguientes métodos:

a) El método hipotenusa, que toma dos argumentos de punto flotante con doble precisión, llamados lado1 y lado2, y que devuelve un resultado de punto flotante, con doble precisión.

b) El método menor, que toma tres enteros (x, y, z) y devuelve un entero.

c) El método instrucciones, que no toma argumentos y no devuelve ningún valor.

d) El método intAFloat, que toma un argumento entero llamado numero y devuelve un resultado de punto flotante (float).

T14. Encuentre el error en cada uno de los siguientes segmentos de programas. Explique cómo se puede corregir.

a)

```
void g() {  
    System.out.println("Dentro del método g");  
}
```



UNIDAD 4. Programación Modular, Clases y Objetos.

```
void h() {  
    System.out.println("Dentro del método h");  
}  
}
```

b)

```
int suma(int x, int y) {  
    int resultado;  
    resultado = x + y;  
}
```

c)

```
void f(float a); {  
    float a;  
    System.out.println(a);  
}
```

d)

```
void producto() {  
    int a = 6, b = 5, c = 4, resultado;  
    resultado = a * b * c;  
    System.out.printf("El resultado es %d\n", resultado);  
    return resultado;  
}
```

e)

```
void tercio(double a, double b) {  
    a = 0.33;  
    return a * b;  
}
```

T15. Declare el método **volumenEsfera()** para calcular y mostrar el volumen de la esfera. Utilice la siguiente instrucción para calcular el volumen:

```
double volumen = (4.0 / 3.0) * Math.PI * Math.pow(radio, 3);
```

Escriba una aplicación en Java que pida al usuario el radio double de una esfera, que llame a **volumenEsfera()** para calcular el volumen y muestre el resultado en pantalla.



UNIDAD 4. Programación Modular, Clases y Objetos.

E2. Queremos hacer una clase que encapsule el nombre y la edad de un empleado que siempre debe ser mayor o igual que 0. Necesitamos conseguir que la edad sea obligatoria. Analiza la siguiente definición de la clase y descubre y soluciona los problemas que pueda tener (A parte de errores léxicos, quizás proporcionando un constructor y cambiando algo...).

```
// Fichero Empleado.java
Public class empleado {
    String nombre;
    int edad;
}

// Fichero PruebaEmpleados.java
public class PruebaEmpleados {
    public static void main(String[] args) {
        Empleado e1 = new Empleado();
        System.out.println( e1.edad );
        e1.edad = 14;
    }
}
```

E3. A la clase Empleados del ejercicio E2 queremos añadirle un método estático (ya que hace lo mismo para todos los objetos) que devuelva el nombre del empleado, algo así:

```
public static String getNombre() { return this.nombre; }
```

Analiza la solución propuesta y soluciona el problema.

E4. A la clase Empleados del ejercicio E2 queremos añadirle el dato sueldo y un password común para que lo usen en el departamento de recursos humanos. Necesitamos poder consultar todos los datos de un empleado desde fuera de la clase y también modificarlos, pero seguir asegurando que la edad de cada uno es mayor o igual a 18 y que el sueldo solamente lo podrán consultar/cambiar si se conoce un password común que compartan todos los objetos. Piensa en una forma



UNIDAD 4. Programación Modular, Clases y Objetos.

de conseguir todo esto.

E5. Necesitamos una clase para manejar puntos en 2 dimensiones. Cada punto queda definido por su coordenada x y por su coordenada y. Creamos la clase con un constructor al que pasamos los dos valores. Analiza el código y encuentra y soluciona los errores.

```
// Fichero Punto2D.java
Public class empleado {
    public double x, y;

    public void Punto2D(double x, double y) {
        x = x;
        y = y;
    }
}

// Fichero PruebaEmpleados.java
public class PruebaPuntos {
    public static void main(String[] args) {
        Punto2D p1 = new Punto2D(3, 2.5);
        System.out.println( p1.x, p2.y );
    }
}
```

E6. Queremos que al usar el constructor por defecto (sin argumentos) se cree el punto (0,0). Además preferimos no añadir más sentencias innecesarias a las que necesitamos. ¿Qué tal te parecen estas soluciones sabiendo que `public Punto2D() { x = 0; y = 0; }` está descartada porque añadimos sentencias?

a) No hacemos nada y cuando se use `new Punto2D();` Java usará el constructor de `Object`, que crea el objeto e inicializa las variables miembro x, y a 0 al no ser locales. Razona la respuesta y luego la confirmas escribiendo código que lo pruebe.

b) `public Punto2D Punto2D() { return new Punto2D(0, 0); }`



UNIDAD 4. Programación Modular, Clases y Objetos.

```
c) public Punto2D() { Punto2D this = new Punto2D(0,0); }
```

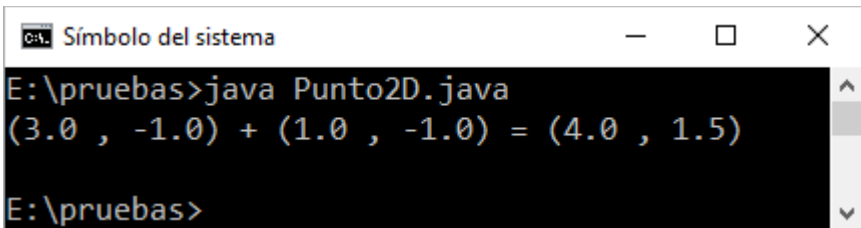
Si no te cuadra ninguna de las anteriores, piensa como hacer el constructor usando el que ya tienes.

E7. Queremos añadir a la clase `Punto2D` getters y setters y una operación suma de forma que reciba de argumento un `Punto2D` y devuelva un nuevo `Punto2D` con la suma de ambos:

$$(x1, y1) + (x2, y2) = (x1 + x2, y1 + y2)$$

Crea un método `main` para probar la clase y completa las siguientes sentencias hasta conseguir una salida como esta:

```
Punto2D p1 = new Punto2D(3, 2.5);
Punto2D p2 = new Punto2D(1, -1);
Punto2D p3 = p1.suma( p2 );
System.out.println( /* ... completa ... */ );
```



```
Símbolo del sistema
E:\pruebas>java Punto2D.java
(3.0 , -1.0) + (1.0 , -1.0) = (4.0 , 1.5)
E:\pruebas>
```

E8. Nos damos cuenta que si queremos imprimir muchos puntos, ahora mismo es un poco lioso. ¿Cómo conseguir que al pasar un punto se imprima "(x, y)"? Sabes que `Object`, como buen padre, presta sus métodos incluido `toString()` a todos sus hijos, incluido `Punto2D`. Por tanto, cambiamos la última línea del código de prueba anterior por:

```
System.out.println( p1 + " + " + p2 + " = " + p3);
```



UNIDAD 4. Programación Modular, Clases y Objetos.

Si no obtienes la salida que querías, quizás tengas que sobrescribir (sustituir, *override* en inglés) el método `toString()` de `Object`.

E9. Ahora necesitas saber si dos `Punto2D` son iguales para poder compararlos:

```
p2.setX( p1.getX() );  
p2.setY( p1.getY() );  
System.out.println(p1 + " es igual a " + p2 + "? " + (p1 == p2) );
```

Como son objetos, supongo que a estas alturas ni se te pasará por la cabeza usar el operador `==` de arriba (solamente te serviría para saber si son el mismo objeto, en cuyo caso sí serían iguales).

Sobreescribe el método `equals()` de `Object` que permita comparar `Puntos2D` con el resto de objetos.

E10. Utiliza la anotación **@Override** delante del método **equals()** de `Punto2D` para comprobar si has realizado bien la sustitución del método del padre, o bien si al equivocarte lo que has hecho es sobrecargar el nombre del método (ligadura dinámica) en vez de sobrescribirlo. Si lo has sobreescrito mal, debes corregirlo. Queremos sustituirlo, no sobrecargar el método **equals()**. Prueba el método corregido con el código de prueba correcto.

E11. Copia el `main` de la clase `Punto2D` a una clase pública llamada `U04EA11` y haz que `Punto2D` no sea pública. Guarda los cambios en el fichero `.java` con el mismo nombre que la clase pública. Añade estas dos nuevas líneas de código a la clase de prueba de `Punto2D`:

```
// Sentencia 1  
if( p1.x >= 0 && p2.y >= 0 ) {  
    System.out.println( p1 + " en primer cuadrante" );  
}  
// Sentencia 2 equivalente
```




UNIDAD 4. Programación Modular, Clases y Objetos.

```
if( p1.getX() >= 0 && p2.getY() >= 0) {  
    System.out.println( p1 + " en primer cuadrante" );  
}
```

Ahora hemos descubierto que sería ventajoso usar internamente un array para almacenar las coordenadas x e y de cada Punto2D, la x en la posición 0, la y en la posición 1 del array. Así podríamos fácilmente crear puntos 3D, 4D, 1D... Cambia el código de la clase Punto2D y sustituye la línea: `public double x, y;` por `double[] c;` Adapta el resto de código de la clase a esta nueva definición y responde (de forma razonada y probando si es necesario) a estas preguntas:

- ¿A cuál de las dos sentencias (1 y 2) afectará el cambio?
- ¿Es bueno usar los getters y setters de los objetos una clase en vez de acceder directamente a sus variables miembro? Responde V (verdadero) y F (falso):
 - Así la clase puede implementar reglas de negocio (restricciones lógicas) y forzar su cumplimiento: ____.
 - Así los cambios internos en cada clase no afectan al código que los usa si la interfaz no cambia: ____.
 - Asegurar los dos puntos anteriores exige que la clase oculte (proteja al máximo) las variables miembro con los modificadores de acceso adecuados): ____.
 - Lo razonable (salvo que existan causas que lo justifiquen) es declarar variables miembro como `private` y métodos de uso externo como `public`: ____.

E12. Capitalizar un string significa convertir las primeras letras de cada palabra del string a mayúscula (si no lo está ya). Por ejemplo, "Ha llegado el momento de la verdad" se convierte en "Ha Llegado El Momento De La Verdad". Haz una subrutina que capitalice un string



UNIDAD 4. Programación Modular, Clases y Objetos.

que se le pasa como parámetro (si observas capitaliza la primera letra de cada frase/primer de cada palabra). Prueba la subrutina en una rutina main() que:

Nota: usa la clase `StringBuilder` para hacerlo de forma eficiente y las funciones `Character.isLetter(char)` y `Character.toUpperCase(char)`.

a) Obtenga la cadena preguntándola al usuario.

```
Símbolo del sistema
E:\pruebas>java U04EA12A.java
Teclee una frase: el amazonas.el mayor rio del mundo
Capitalizada: El Amazonas.El Mayor Río Del Mundo
E:\pruebas>_
```

b) Obtenga la cadena como parámetro del programa.

```
Símbolo del sistema
E:\pruebas>java U04EA12B.java
Debe pasar la frase a capitalizar

E:\pruebas>java U04EA12B.java vini, vidi, vinchi
Capitalizada: Vini, Vidi, Vinchi
E:\pruebas>_
```

c) Si no tiene parámetro, pregunte al usuario la cadena, si tiene parámetro, lo capitalice.

E13. Los dígitos hexadecimales son los decimales ('0' a '9') y las letras de la 'A' a la 'F'. En el sistema de numeración hexadecimal los dígitos representan valores de 0 a 15. Escribe una función llamada **hexValor()** que usando una sentencia **switch**, encuentre el valor hexadecimal de un carácter que se pasa como parámetro (debe dar igual mayúsculas o



UNIDAD 4. Programación Modular, Clases y Objetos.

minúsculas, 'f' y 'F' valen 15). Si el carácter que se pasa no está entre '0' y 'F' devuelve -1.

Una cifra en hexadecimal es una cadena con dígitos hexadecimales, como 34A7, ff8, 174204 o FADE. Si s es una cadena que contiene un número expresado en hexadecimal, entonces su valor en decimal se puede calcular así:

```
valor = 0;
for ( i = 0; i < s.length(); i++ )
    valor = valor * 16 + hexValor( s.charAt(i) );
```

Haz un programa que acepte como parámetro una cadena hexadecimal, y si no se la pasan que la pida al usuario y muestre el valor decimal si es válido o un mensaje de error si tiene caracteres no hexadecimales.

E14. Escribe una función que simule el lanzamiento de un par de dados hasta que salga el número que pasas como parámetro (debe estar entre 2 y 12). La función devuelve la cantidad de tiradas que ha costado sacar ese número.

a) Usa una aserción para controlar la precondition y la postcondición (lo que haya) con un mensaje personalizado indicando la función donde se produce el error en el mensaje.

b) Ahora sustituye la aserción por una una excepción de parámetros para asegurarte de que la función funcionará.

E15. Escribe un método que acepte un número arbitrario de argumentos enteros, (como mínimo 2) y devuelva el máximo.

E16. Para redondear números a lugares decimales específicos, use una instrucción como la siguiente:

```
y = Math.floor(x * 10 + 0.5) / 10;
```

que redondea x en la posición de las décimas (es decir, la primera



UNIDAD 4. Programación Modular, Clases y Objetos.

posición a la derecha del punto decimal), o:

```
y = Math.floor(x * 100 + 0.5) / 100;
```

que redondea x en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba una aplicación que defina cuatro métodos para redondear un número x en varias formas:

redondearA(x , decimales)

que redondea x en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba una aplicación que defina un método para redondear un número x de varias formas:

E17. Se dice que un número entero es un *número perfecto* si sus factores, incluyendo el 1 (pero no el propio número), al sumarse dan como resultado el número entero. Por ejemplo, 6 es un número perfecto ya que $6 = 1 + 2 + 3$.

Escriba un método llamado **esPerfecto()** que determine si el parámetro **numero** es un número perfecto. Use este método en una aplicación que muestre todos los números perfectos entre 1 y 1,000. Muestre en pantalla los factores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora: evalúe números más grandes que 1,000. Muestre los resultados.

E18. Se dice que un entero positivo es *primo* si puede dividirse de forma exacta solamente por 1 y por sí mismo. Por ejemplo: 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no. Por definición, el número 1 no es primo.

- Escriba un método que determine si un número es primo.
- Use este método en una aplicación que determine y muestre en pantalla todos los números primos menores que 10,000. ¿Cuántos números hasta 10,000 tiene que probar para



UNIDAD 4. Programación Modular, Clases y Objetos.

asegurarse de encontrar todos los números primos?

c) Al principio podría pensarse que $n/2$ es el límite superior para evaluar si un número n es primo, pero sólo es necesario ir hasta la raíz cuadrada de n . Vuelva a escribir el programa y ejecútelo de ambas formas.

E19. Intente aplicar el diseño top-down a este problema e identificar las subrutinas que sería conveniente implementar. Escriba una aplicación que muestre un menú con 3 opciones:

```
Menu
====

1. Lanzar moneda.
2. Ver resultados.
3. Salir.

Escoja una opción (1-3):
```

El programa muestra repetidamente el menú hasta que la opción elegida sea alguna de la 1 a la 3. Si es la '3', el programa acaba. Si es la '1' (Lanzar moneda), de forma aleatoria muestra si ha salido cara o cruz y vuelve a mostrar el menú. Cada vez que se lanza la moneda, cuenta el número de veces que aparezca cada uno de los lados de la moneda. Si elige la opción '2' se muestra en pantalla qué porcentaje de caras y de cruces han salido y vuelve a mostrar el menú.

E20. Crea una clase llamada **CuentaBancaria** que tendrá los siguientes atributos:

- Titular y saldo (puede tener decimales). El titular será obligatorio y la cantidad es opcional.
- Crea dos constructores que cumplan lo anterior.
- Crea sus métodos get, set y toString.



UNIDAD 4. Programación Modular, Clases y Objetos.

- Tendrá dos métodos especiales:
 - `ingresar(double cantidad)`: se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
 - `retirar(double cantidad)`: se retira una cantidad de la cuenta, si restando la cantidad actual a la que nos pasan es negativa, se lanza una excepción.

E21. Haz una clase llamada **Cliente** que utiliza una clínica de una dietista y que siga las siguientes condiciones:

- Sus atributos son: nombre, edad, `numPaciente`, sexo (H hombre, M mujer), peso y altura. No queremos que se accedan directamente a ellos, así que piensa que modificador de acceso es el más adecuado. Por defecto, todos los atributos menos el `numPaciente` serán valores por defecto según su tipo. Sexo sera hombre por defecto, usa una constante para ello.
- Se necesitan varios constructores: Un constructor por defecto. Un constructor con el nombre, edad y sexo. Un constructor con todos los atributos.
- Los métodos que se implementaran son:
 - **`calcularIMC()`**: calcula si la persona esta en su peso ideal (peso en kg/(altura² en m)), si esta fórmula devuelve un valor menor que 20, la función devuelve un -1, si devuelve un número entre 20 y 25 (incluidos), significa que esta por debajo de su peso ideal y la función devuelve un 0 y si devuelve un valor mayor que 25 significa que está por encima de su peso ideal y devuelve un +1.
 - **`MayorDeEdad()`**: indica si es mayor de edad, devuelve un booleano.
 - **`comprobarSexo(char sexo)`**: comprueba que el sexo



UNIDAD 4. Programación Modular, Clases y Objetos.

introducido es correcto. Si no es correcto, sera H. No será visible al exterior.

- **toString():** devuelve toda la información del objeto.
- **generaNumPaciente():** genera un número de paciente usando un valor estático que cuenta los objetos de la clase, inicialmente a 0. Genera a partir de este su número de paciente. Este método sera invocado cuando se construya el objeto. No será visible al exterior.
- Métodos set de cada parámetro, excepto de numPaciente.

Ahora, crea una clase ejecutable que haga lo siguiente: Pide por teclado el nombre, la edad, sexo, peso y altura. Crea 3 objetos de la clase anterior, el primer objeto obtendrá las anteriores variables pedidas por teclado, el segundo objeto obtendrá todos los anteriores menos el peso y la altura y el último por defecto, para este último utiliza los métodos set para darle a los atributos un valor.

Para cada objeto, deberá comprobar si esta en su peso ideal, tiene sobrepeso o por debajo de su peso ideal con un mensaje. Indicar para cada objeto si es mayor de edad. Por último, mostrar la información de cada objeto. Puedes usar métodos en la clase ejecutable.

E22. Vamos a realizar una clase llamada ConjuntoEnteros que define el comportamiento de un conjunto de números enteros. Sus objetos deben ser inmutables.

- Puede tener de 0 a infinitos elementos.
- No tiene elementos repetidos.

Tienen:

- Un nombre
- Un array de elementos enteros.
- Constructores: por defecto y al que se le pasa un array.
- Operaciones públicas: boolean esVacio(), int cardinal(), void



UNIDAD 4. Programación Modular, Clases y Objetos.

insertaElemento(int), borraElemento(int), boolean
pertenece(int), ConjuntoEntero union(ConjuntoEntero c).

- Implementa: toString() y equals().