Programación 1ºDAM/DAW

Introducción

Material didáctico del profesor Javier AR para el módulo de Programación de 1ºDAM/DAW, adaptado a normativa del curso 23/24.

El principal lenguaje de programación empleado en este manual es Java.

UT01 - Introducción a la programación

UT02 - Identificación de los elementos de un programa informático

Imprimiendo en Java

Contenido

En este ejemplo vamos a entender cómo imprimir por la pantalla. Esta es una herramienta esencial que necesitamos para poder hacer pruebas, comprobar que nuestro código hace lo que esperábamos y como herramienta primitiva de depuración.

Practicaremos con:
- System.out.println
- System.out.print
- Uso de colores y caracteres especiales en cadenas de texto (ascii/unicode)

Además, repasamos los comentarios; con los dos tipos de notaciones, de una línea y de múltiples líneas.

Código de ejemplo

```
public class UT02E01Imprimiendo {

    public static void main(String[] args) {
        // Texto + nueva línea
        System.out.println("-----Imprimiendo textos-----");
        System.out.println("Esto imprime un texto.");
        System.out.println("Esto imprime otro texto.");

        // Texto sin nueva línea
        System.out.println("-----Imprimiendo sin cambio de línea-----");
        System.out.print("Uno");
        System.out.print("Dos");
        System.out.print("Tres");
        System.out.print("\n");

        // Añadiendo colores (ansi)
        System.out.println("-----Imprimiendo con colores-----");
```

```
        System.out.println("\033[30m colores");
        System.out.println("\033[31m colores");
        System.out.println("\033[32m colores");
        System.out.println("\033[33m colores");
        System.out.println("\033[34m colores");
        System.out.println("\033[35m colores");
        System.out.println("\033[36m colores");
        System.out.println("\033[37m colores");

System.out.println("\033[31mc\033[32mo\033[33ml\033[34mo\033[35mr\033[36me\033[37ms");

        // Caracteres especiales (unicode)
        System.out.println("-----Imprimiendo caracteres especiales-----");
        System.out.println("\u263A");
        System.out.println("\u00A9");
        System.out.println("\u24FE");

    }

}
```

Ejercicios

Ejercicio 1.
Dibuja un tablero de 3 en raya en el que se aprecien las fichas (X O)y los bordes de las casillas.

Ejercicio 2.
Dibuja esta figura por pantalla:

```
@@@@@@@@
@ x  x @
@  uu  @
@@@@@@@@
```

Ejercicio 3.
Ejecuta las siguientes líneas de código en un nuevo programa:

```
System.out.print("\033[35m uno");
System.out.print("dos");
```

Reemplaza "print" por "println" ¿Qué diferencias aprecias?

Referencias y ampliación

Variables

Contenido

En este ejemplo vamos a aprender a usar variables. Las variables son contenedores de datos de algún tipo. Existen diferentes tipos de datos, en este ejemplo veremos cómo se declaran, cómo se le asignan datos, cómo se reeemplazan los datos con otros nuevos y cómo podemos emplear las variables en el resto del programa.

Java es un lenguaje de tipado fuerte, esto quiere decir que todas las variables que usemos en nuestro programa tienen que estar definidas y deben ser del mismo tipo de dato que el valor que estamos asignado.

El formato recomendado para poner nombre a las variables se conoce como "lowerCamelCase" donde la primera palabra comienza por minúscula y las demás se escriben juntas pero con la primera letra mayúscula. Se evita poner la primera letra en mayúscula para no confundirla con el nombre de una clase, en la que sí usamos mayúscula en la primera palabra. Esto es solo un consejo de estilo, los programas no fallan si no se cumple, pero es lo más adecuado para que tanto nosotros, como los demás desarrolladores que lean nuestro código lo entiendan de forma sencilla.

Los nombres de las variables no pueden tener símbolos ($%#@-+). Pueden tener números, pero nunca al inicio.

Código de ejemplo

```
public class UT02E02Variables {

    public static void main(String[] args) {

        // Ejemplo de variable de tipo entero
        // números son decimales con 4 bytes
        System.out.println("-----Variable de tipo entero-----");
```

```
        int x;
        x = 1;
        System.out.println("x vale: " + x);
        x = 2;
        System.out.println("x ahora vale: " + x);

        // Más variables numéricas
        // en este caso declaramos y damos valor en la misma línea
        System.out.println("-----Más variables numéricas-----");
        byte vbleByte = 100;
        short vbleShort = 5000;
        long vbleLong = 1000000000;
        System.out.println(vbleByte);
        System.out.println(vbleShort);
        System.out.println(vbleLong);

        // Podemos declarar varias variables del mismo tipo en una
sentencia
        System.out.println("-----Declarando variables en una
sentencia-----");
        int a, b, c;
        a = 1;
        b = a;
        c=b; // los espacios no son necesarios pero aportan claridad
        System.out.println("a: " + a);
        System.out.println("b: " + b);
        System.out.println("c: "+c); // los espacios no son necesarios

        // Variables numéricas con decimales
        System.out.println("-----Variables con decimales-----");
        float vbleFloat = 1.34f;
        double vbleDouble = 3.78d;
        System.out.println(vbleFloat);
        System.out.println(vbleDouble);

        // Variables booleanas (almacenan verdadero o falso)
        System.out.println("-----Variables booleanes true/false-----");
        boolean vbleBoolean = true;
        System.out.println("Esto es " + vbleBoolean);
        vbleBoolean = false;
        System.out.println("Esto es " + vbleBoolean);

        // Variables que almacenan caracteres
```

```
        System.out.println("-----Variables de caracteres-----");
        char vbleChar = 'a'; // se usan comillas simples
        System.out.println("La letra es: " + vbleChar);
        vbleChar = 100; // acepta el valor numérico ASCII
        System.out.println("La letra es: " + vbleChar);

        // El tipo de datos String no es un tipo primitivo
        // es un tipo muy útil que nos permite almacenar una
        // cadena de texto.
        System.out.println("-----Variables de cadena de texto-----");
        String saludo = "¡Hola a todos!";
        System.out.println(saludo);

        // Los tipos primitivos empiezan por letra minúscula, mientras
        // que los tipos no primitivos empiezan por mayúscula.

        // Si no queremos que una variable pueda cambiar su valor,
        // podemos hacerlo usando la palabra reservada final
        System.out.println("-----Variables que no pueden cambiar-----");
        final int descuento = 10;
        // descuento = 20; produciría un error!
        System.out.println("El descuento es " + descuento);
    }

}
```

Ejercicios

Ejercicio 1.
Echa un vistazo a los tipos de datos primitivos:
https://www.w3schools.com/java/java_data_types.asp
Observa el rango numérico de cada tipo de dato.

Ejercicio 2.
Crea un programa con dos variables de distinto tipo, intenta asignar la primera variable a la segunda. Prueba con diferentes tipos y familiarízate con los errores que genera.

Ejercicio 3.
Crea tres variables de tipo char y emplearlas en una sola sentencia System.out.println para generar una palabra de tres letras como Sol.

Ejercicio 4.
Imprime por pantalla una dirección postal tal y como aparecería en una carta. Todos los campos deben ser variables.

Referencias y ampliación

Operadores

Contenido

En este ejemplo vamos a aprender a usar los operadores. Los operadores nos permiten actuar con los datos y las variables para realizar operaciones y cálculos que necesitemos en nuestros programas.

En este ejemplo estudiaremos los operadores aritméticos, los de asignación, los de comparación y los lógicos.

Código de ejemplo

```
public class UT02E03Operadores {

   public static void main(String[] args) {

       // Operadores aritméticos
       System.out.println("-----Operadores aritméticos-----");
       int a = 5, b = 2, resultado;
       resultado = a + b;
       System.out.println("Suma: " + resultado);
       resultado = a - b;
       System.out.println("Resta: " + resultado);
       resultado = a * b;
       System.out.println("Multiplicación: " + resultado);
       resultado = a / b;
       System.out.println("División: " + resultado);
       resultado = a % b;
       System.out.println("Módulo: " + resultado);
       resultado = a++;
       System.out.println("Incremento++: " + resultado);
       System.out.println("Valor de a: " + a);
```

```
resultado = b--;
System.out.println("Decremento--: " + resultado);
System.out.println("Valor de b: " + b);
a = 5;
b = 2;
resultado = ++a;
System.out.println("++Incremento: " + resultado);
System.out.println("Valor de a: " + a);
resultado = --b;
System.out.println("--Decremento: " + resultado);
System.out.println("Valor de b: " + b);
a++; // Esta expresión suele usarse sola sin asignación.


// Operadores de asignación
System.out.println("-----Operadores de asignación-----");
int x;
x = 1;
x += 1; // x = x + 1
x -= 1;
x *= 1;
x /= 1;
x %= 1;

// Operadores de comparación (devuelven true o false)
System.out.println("-----Operadores de comparación-----");
a = 2;
b = 1;
boolean r;
System.out.println("a=" + a + " b=" + b);
r = (a == b);
System.out.println("(a == b): " + r); // igual
r = (a != b);
System.out.println("(a != b): " + r); // no igual
r = (a > b);
System.out.println("(a > b): " + r); // mayor
r = (a < b);
System.out.println("(a < b): " + r); // menor
r = (a >= b);
System.out.println("(a >= b): " + r); // mayor o igual
r = (a <= b);
System.out.println("(a <= b): " + r); // menor o igual
```

```
        // Operadores lógicos (AND, OR y NOT)
        System.out.println("-----Operadores lógicos-----");
        a = 3;
        b = 7;
        System.out.println("a=" + a + " b=" + b);
        r = (a < 5 && b > 5); // AND - es cierto si las dos partes lo
son
        System.out.println("(a < 5 && b > 5): " + r);
        r = (a < 5 || b > 10); // OR - es cierto si una de las dos lo es
        System.out.println("(a < 5 && b > 5): " + r);
        r = !r; // Invierte el resultado lógico false->true true->false
        System.out.println("Inverso del anterior: " + r);
    }

}
```

Ejercicios

Ejercicio 1.
Crea un programa en java que genera la tabla de multiplicar de un número que se declara en una variable al inicio del mismo. Solo se puede usar una variable de tipo entero en todo el programa.

Ejercicio 2.
Crea un programa que a partir de la fecha de nacimiento diga la edad en años, meses y días de la persona.
¿Y los bisiestos? ¡Es muy tedioso! Investiga "java local date", e intenta resolver el ejercicio haciendo uso de ello.

Ejercicio 3.
Crea un programa que saque por pantalla las calificaciones obtenidas en 4 asignaturas, y debe indicar de alguna forma cual de ellas tiene la mayor calificación.

Ejercicio 4.
Crea un programa que calcule la hipotenusa de un triángulo rectángulo a partir del tamaño de sus catetos, los valores deben expresarse con decimales.

Referencias y ampliación

Conversión de tipos

Contenido

En este ejemplo vamos a aprender a convertir tipos de datos compatibles. En ocasiones tendremos un un dato en un tipo de variable, que es compatible con otro tipo de datos y necesitaremos cambiar su tipo.

Por ejemplo, en una caja de zapatos podremos guardar zapatos, pero podemos guardar unas unas gafas. Teniendo dentro unas gafas, podemos intentar guardarlas en una funda para gafas y funcionará, pero al intentar guardas unos zapatos en la funda para gafas no podremos.

Tenemos dos formas de convertir de tipos:
- De forma implícita: En esta forma, algunos tipos de datos pueden convertirse a otros de mayor tamaño.
  byte -> short -> char -> int -> long -> float -> double
- De forma explícita: cuando queremos convertir tipos de mayor tamaño a otros de menor tamaño.
 double -> float -> long -> int -> char -> short -> byte

En este ejemplo, adicionalmente, veremos un tipo de conversión especial que nos será útil en el trabajo de las primeras unidades del curso, la conversión de texto a un tipo numérico.

Código de ejemplo

```
public class UT02E04ConversionesDeTipos {

   public static void main(String[] args) {
        // Conversión implícita
        System.out.println("-----Conversión implícita-----");
        int vbleTipoInt = 5;
        double vbleTipoDouble;
        vbleTipoDouble = vbleTipoInt;
        System.out.println("Implícita: Valor de int: " + vbleTipoInt);
        System.out.println("Implícita: Valor de double: " +
vbleTipoDouble);

        byte vbleTipoByte = 10;
        short vbleTipoShort;
        vbleTipoShort = vbleTipoByte;
```

```
        System.out.println("Implícita: Valor de byte: " + vbleTipoByte);
        System.out.println("Implícita: Valor de short: " +
vbleTipoShort);

        // Conversión explícita
        System.out.println("-----Conversión explícita-----");
        double vbleTipoDouble2 = 3.2;
        int vbleTipoInt2;
        vbleTipoInt2 = (int)vbleTipoDouble2;
        System.out.println("Explícita: Valor de double: " +
vbleTipoDouble2);
        System.out.println("Explícita: Valor de int: " + vbleTipoInt2);

        // Convirtiendo texto a número
        System.out.println("-----Conversión texto a número-----");
        String cadena1 = "7";
        String cadena2 = "hola";
        int numero;
        numero = Integer.parseInt(cadena1);
        //numero = Integer.parseInt(cadena2); // Error :(
        System.out.println("El valor de la variable número es: " +
numero);

    }

}
```

Ejercicios

Ejercicio 1.
Intenta hacer más conversiones de datos implícitas. ¿Funcionan todas? ¿Alguna da error?
Si no conoces bien todos los tipos de datos busca en internet sobre ellos así sabrás el formato que usan.

Ejercicio 2.
Realiza diferentes conversiones de datos. Algunas darán error, lo importante de este ejercicio es que te familiarices con esos errores y el texto que los describe.

Ejercicio 3.

Realiza conversiones de texto a variables de tipo double y float. (Busca en internet si no encuentras la forma de hacerlo!)

Ejercicio 4.
Hemos usado esta sentencia para convertir texto a número:
numero = Integer.parseInt(cadena1);
Investiga qué es "Integer".
Realiza un programa que use variables de tipo "Integer" y de tipo "int". ¿En qué se diferencian?

Referencias y ampliación

Lectura del teclado

Contenido

En este ejemplo vamos a leer información desde el teclado que luego podremos emplear en nuestro programa. Existen dos formas de leer información desde el teclado mediante la terminal:

System.console().readLine(). Esta versión no la emplearemos ya que genera errores cuando trabajamos en entornos de desarrollo como NetBeans o Eclipse.

La clase Scanner. Esta clase forma parte de la librería java.util. Para poder usarla en nuestros programas debemos importarla, pero NetBeans nos ayudará.

Código de ejemplo

```
public class UT02E05LecturaDelTeclado {

   public static void main(String[] args) {

       // Lectura con System.console().readLine()
       // En estos comentarios se muesta la forma
       // de hacerlo, pero está todo el código
       // comentado para que generar errores.
       // No es importante saber usarla, ya que
       // usaremos la clase Scanner durante el
       // curso.
```

```
            // String nombre;
            // System.out.print("Introduce tu nombre: ");
            // nombre = System.console().readLine();
            // System.out.println("Hola " + nombre);
        // Si queremos un número debemos recogerlo
        // como String y luego hacer una conversión
        // de tipos.



        // Lectura desde el teclado con la clas Scanner
        Scanner lectorTeclado = new Scanner(System.in);
        // lectorTeclado es una variable de tipo Scanner
        // nextLine
        String localidad, calle;
        System.out.print("¿En qué localidad vives?: ");
        localidad = lectorTeclado.nextLine();
        System.out.print("¿En qué calle vives?: ");
        calle = lectorTeclado.nextLine();  // Podemos reutilizarla
        System.out.println("Localidad: " + localidad);
        System.out.println("Calle: " + calle);

        // next
        String primera, segunda;
        System.out.println("Escribe dos palabras separadas por
espacio:");
        primera = lectorTeclado.next();
        segunda = lectorTeclado.next();
        System.out.println("Primera palabra: " + primera);
        System.out.println("Segunda palabra: " + segunda);

        // nextInt
        int edad;
        System.out.print("¿Cuántos años tienes?: ");
        edad = lectorTeclado.nextInt();
        System.out.println("Tienes " + edad + " años.");

    }

}
```

Ejercicios

Ejercicio 1.
Realiza un programa que solicite dos números al usuario y devuelva la suma, la resta, la multiplicación y la división de esos números, el resultado debe mostrar las operaciones completas (Ejemplo: 7 * 2 = 14)

Ejercicio 2.
Modifica el ejercicio 4 del ejemplo 3, en el que calculamos la hipotenusa de un triángulo, ahora el programa debe solicitar el valor de los catetos al usuario, además debe solicitar las unidades en las que se expresan los valores (centímetros, metros, kilómetros, etc.), y el resultado debe indicar dicha unidad.

Ejercicio 3.
Escribe un programa que a partir del precio de venta y el % de IVA calcule la parte que se queda el comercio y la parte que son impuestos.

Ejercicio 4.
Realiza un programa que a partir de tres números indique cual es el mayor, cuál es el menor o si son iguales.

Ejercicio 5.
Asegúrate que la javadoc de tu JDK está bien configurada en NetBeans.
(Consulta la sección de información de la unidad en el aula virtual).
A partir de este código:

```
String a;
Scanner b;
Integer c;
```

Escribe en una nueva línea "a." y deja que aparezca la ayuda de javadoc.
Explora las distintas opciones que aparecen e intenta deducir para qué se pueden emplear. Haz lo mismo con "b." y "c.".
"Integer" es un tipo similar a "int". Busca información sobre ello en internet e intenta entender qué papel juegan cada uno de estos tipos.

Referencias y ampliación

UT03 - Uso de estructuras de control

If

Contenido

En este ejemplo aprenderemos a usar la sentencia condicional if, junto con las expresiones else y else if.

If es una sentencia condicional que nos va a permitir ejecutar un bloque de código sólo si se cumple determinada condición.

Las sentencias if, valoran esta condición en base a true/false. Podemos indicar este true o false con una variable de tipo boolean o con una expresión que arroje como resultado true/false.
5 + 7 arroja como resultado 12, no es válido
5 == 7 arroja como resultado false, es válido.

Las operaciones que arrojan resultados booleanos son:
 == != < > <= >=

Además, podemos realizar varias operaciones de este tipo con los operadores lógicos:
 && || !

Esta herramienta es muy poderosa y se usa en todos los niveles de la programación y el desarrollo para infinidad de tareas.

Código de ejemplo

```
public class UT03E01If {

    public static void main(String[] args) {
        // Ejemplos de uso de la ejecución condicional if
        System.out.println("-----Sentencia if-----");

        int a = 3;
        if (a < 5) {
            System.out.println("AAA");
```

```
        }

        System.out.println("Esto se ejecuta siempre, no está dentro de
un if.");

        boolean z;
        z = true;
        if (z) {
            System.out.println("BBB");
        }

        int b = 7;
        int c = 6;
        // usamos varias operaciones pero el resultado final será true o
false
        if (!(((a < b) && (c > b)) || (b == a))) {
            System.out.println("CCC");
        }

        z = !(((a < b) && (c > b)) || (b == a));
        if (z) {
            System.out.println("DDD");
        }

        // De esta forma podemos comparar un string
        String cadena = "Hola", cadena2 = "Hola";
        if (cadena.equals(cadena2)) {
            System.out.println("EEE");
        }
        if ("Hola".equals(cadena2)) {
            System.out.println("FFF");
        }
        if (cadena.equals("Hola")) {
            System.out.println("GGG");
        }

        // Ejemplo de uso de la sentencia else asociada a un if
        System.out.println("-----Sentencia else-----");
        a = 2;
        if (a > 5) {
            System.out.println("HHH");
        } else {
            System.out.println("III");
```

```
        }

        // Ejemplo de uso de else if
        System.out.println("-----Sentencia else if-----");
        a = 5;
        if (a == 4) {
            System.out.println("JJJ");
        } else if (a == 5) {
            System.out.println("KKK");
        } else {
            System.out.println("LLL");
        }

    }

}
```

Ejercicios

Ejercicio 1.
Escribe un programa que lea un número por el teclado y diga si el número es par o impar.

Ejercicio 2.
Escribe un programa que solicite una contraseña (con mayúsculas, minúsculas, * signos de puntuación y números), si la contraseña es correcta tendrá acceso al sistema, si es incorrecta no.

Ejercicio 3.
Escribe un programa que solicite la edad del comprador de un billete de autobús y le mostrará el precio del billete. Si es menor de 7 años el precio será gratuito, menor de 26 será un euro, hasta 65 años 2 euros y mayor de 65 gratuito.

Ejercicio 4.
Escribe un programa que calcule el coste de un parking. El programa pedirá las horas y minutos que ha estado estacionado el vehículo.
El precio es:
- 10 céntimos el minuto los primeros 60 minutos
- 7 céntimos el minuto los siguientes 120 minutos
- 5 céntimos los minutos restantes.

Si el tiempo total es mayor a 3000 minutos, en lugar de sacar el precio sacará un mensaje diciendo "El coche se encuentra en el depósito. Recibirá una multa en su domicilio.".

Referencias y ampliación

Switch

Contenido

En este ejemplo aprenderemos a usar la sentencia condicional switch. Esta sentencia no será útil cuando queramos comparar una variable con una serie de valores concretos. Si bien esta operación la podemos realizar con una serie de else if, el uso de switch tiene algunas leves diferencias y puede generar código más limpio.

Código de ejemplo

```
public class UT03E02Switch {

    public static void main(String[] args) {
        // Ejemplo de uso de switch
        System.out.println("-----Ejemplo básico de switch-----");
        int dia = 3;
        String nombreDia;

        switch (dia) {
            case 1:
                nombreDia = "lunes";
                break;
            case 2:
                nombreDia = "martes";
                break;
            case 3:
                nombreDia = "miércoles";
                break;
            case 4:
```

```
                nombreDia = "jueves";
                break;
            case 5:
                nombreDia = "viernes";
                break;
            default:
                nombreDia = "no existe ese día";
        }

            System.out.println("El día seleccionado es: " + nombreDia);


        // Comprobando el uso opcional de break y default
        System.out.println("-----Ejemplo switch con ausencia de
break-----");
        int a = 2, b = 1;
        switch (a) {
            case 1:
                b = 5;
                break;
            case 2:
                b = 10;
            case 3:
                b = 20;
                break;
            default:
                b = 3;
        }
        System.out.println("El valor de b es: " + b);

        System.out.println("-----Ejemplo switch con ausencia de
default-----");
        a = 4;
        b = 1;
        switch (a) {
            case 1:
                b = 5;
                break;
            case 2:
                b = 10;
                break;
            case 3:
                b = 20;
```

```
                break;
        }
        System.out.println("El valor de b es: " + b);
    }

}
```

Ejercicios

Ejercicio 1.
Reescribe el ejercicio del precio del billete del autobús de (UT03E01if) empleando la instrucción switch.

Ejercicio 2.
Escribe un programa que genere un menú de opciones para el usuario. El usuario debe pulsar una de las opciones y el programa responder con la opción pulsada.

Ejercicio 3.
Escribe un programa que pregunte por el día de la semana que quieres reservar pista de tenis. Tras seleccionar el día debe decir que pista quiere reservar, y si quiere contratar luz artificial.
Las pistas disponibles por días son:
Lunes (1, 2 y 3), Martes (4, 5 y 6) y Miércoles (1, 4 y 7).
La luz se puede contratar todos los días.
El programa debe mostrar la secuencia de operaciones del usuario y guiarle, por ejemplo si decide reservar martes, el programa debe decir qué pistas están libres ese día para que él pueda decidir.
Al final debe mostrarse el resultado de la reserva.
Ejemplo: Ha reservado la pista 4 el miércoles sin luz artificial.

Ejercicio 4.
Escribe un programa que a partir de tres números introducidos por el teclado, los muestre ordenados de mayor a menor.

Ejercicio 5.
Escribe un programa que salude al usuario, pero el usuario debe poder seleccionar en qué color quiere ser saludado.

Referencias y ampliación

For

Contenido

En este ejemplo aprenderemos a usar la instrucción iterativa (for). Esta instrucción nos permite ejecutar un grupo de instrucciones, de forma repetida, un número determinado de veces.
Por ejemplo, necesitamos hacer un programa que cuenta hasta 100. Podríamos escribir 100 sentencias System.out.println, pero sería muy tedioso y repetitivo.
La sentencia for nos va a permitir escribir el System.out.println una sola vez e indicar que se ejecute las 100 veces.

Código de ejemplo

```
public class UT03E03For {

    public static void main(String[] args) {
        // Ejemplo básico de bucle for
        System.out.println("-----Ejemplo básico de for-----");
        for (int i = 1; i < 5; i++) {
            System.out.println(i);
        }

        // Observa el interior del paréntesis tras el for
        // tiene esta estructura:
        // for (cosa1; cosa2; cosa3;)
        // ¿Qué es cada cosa?
        // - cosa1: Es una instrucción de código que se
        //          ejecutará antes de comenzar el bucle
        // - cosa2: es la condición lógica (true o false)
        //          que se evaluará en cada iteración del
        //          bucle, si es true se ejecuta de nuevo,
        //          si es false termina el bucle.
        // - cosa3: es una instrucción que se ejecuta al teminar
        //          cada ejecución del bloque de código que hay
        //          dentro de las llaves {} del for.

        // Entonces, nuestro ejemplo anterior, declara una variable
        // entera llamada i con valor 1, luego evalua la exprexión
```

```
        // lógica i <5, como esta condición es true, entra en las {}
        // y ejecuta todo su contenido, al finalizar la ejecución del
        // contenido de las {} ejecuta i++. En ese momento vuelve a
        // evaluar la expresión lógica i < 5, cuando esta condición
        // sea true, entrará de nuevo en las llaves {}, fuando sea
        // false terminará el bucle.

        // Ejemplo poco ortodoxo de for
        System.out.println("-----Ejemplo poco ortodoxo de for-----");
        int p = 3;
        for(System.out.println("Yo me llamo Ralph"); p != 37; p = 37) {
            System.out.println("¡Corre plátano!");
        }

        // No debes ceñirte al uso del for del primer ejemplo
        // es una herramienta muy potente que puede dar rienda
        // suelta a la creatividad.
        // Aprende de estos ejemplos.
        System.out.println("-----Lanzando un cohete-----");
        for (int i = 5; i != 0; i--) {
            System.out.println("Despegue en " + i);
        }

        System.out.println("-----Hmmm....-----");
        for (int i = 33; i < 43; i=i+2){
            for (int j = 0; j < 8; j++){
                System.out.print("\033[3" + j + "m" +
Character.toString(j + i));
            }
            System.out.print("\n");
        }

        // Las variables i que estamos declarando dentro de los
paréntesis
        // de los for, "nacen" y "mueren" en cada for. No se pueden usar
        // fuera de ellos. Más adelante en el curso se explicará esto
        // con más detalle.
    }

}
```

Ejercicios

Ejercicio 1.
Escribe de nuevo el ejercicio de calcular una tabla de multiplicar empleando un bucle for.

Ejercicio 2.
Escribe un programa que recorra con un bucle los números del 1 al 100 y saque por pantalla los que sean múltiplo de 7 de la siguiente forma:
Los números múltiplos de 7 son: 7, 14, 21...

Ejercicio 3.
¿Recuerdas qué pasaba si escribes un char en un System.out.print? ¡Bien! Me alegro de que lo recuerdes.
Crea un programa en el que declares esta línea:
String frase = "Zubat es mejor que Charmander.";
Luego escribe frase.charAt... y familiarízate con ese método, ¿qué hace?, ¿qué parámetros recibe?, ¿qué parámetros devuelve?
¡Bien! ¡Bien!, vas bien.
En el programa debes calcular la suma de todos los caracteres de la frase.

Ejercicio 4.
Realiza un programa que permita hacer log in. Para ello el usuario debe introducir una contraseña. Si la acierta, entrará, si la falla debe generar un mensaje diciendo que le quedan X intentos, si la falla en todos los intentos debe decir que la cuenta ha quedado bloqueada.

Ejercicio 5.
Escribe un programa que saque por pantalla un cuadrado, el tamaño del cuadrado y el carácter empleado para dibujarlo debe ser introducido por el usuario.
Ejemplo de ejecución:
Tamaño del cuadrado: 4
Carácter empleado: @
Aquí tiene su cuadrado:
@@@@
@ @
@ @
@@@@
Si quieres puedes ampliarlo pidiendo un color y un carácter para el relleno. Si quieres seguir ampliándolo, modifícalo para que la figura pueda ser un triángulo, o un círculo.

Referencias y ampliación

While y do while

Contenido

En este ejemplo aprenderemos a usar la intrucción iterativa (while).

Esta instrucción nos permite ejecutar un grupo de instrucciones, de forma repetida, mientras una condición lógica (true/false) tenga el valor true.

Por ejemplo, queremos hacer una llamada telefónica para pedir una cita médica. La condición que nos interesa cumplir es la de ser atendidos. Si no nos atienden por teléfono volveremos a llamar hasta que lo consigamos. Así, repetiremos la llamada tantas veces como sea necesario hasta que finalmente consigamos la cita, en ese momento dejaremos de hacer llamadas al centro de salud.

El bucle do…while es muy similar, pero tiene una estructura ligeramente distinta. Este bucle, nos asegura que, al menos, su contenido se ejecutará una vez, independientemente del valor de la condición lógica, para el resto de iteraciones se evaluará la condición del mismo modo que en un bucle while.

Código de ejemplo

```
public class UT03E04WhileDoWhile {

   public static void main(String[] args) {

        // Ejemplo básico de bucle while
        System.out.println("-----Ejemplo básico de while---");
        int i = 0;
        while (i < 5) {
            System.out.println(i);
            i++;
        }
```

```
        // Otro ejemplo de while
        i = 0;
        System.out.println("-----Ejemplo flag-----");
        boolean flag = true;
        while(flag) {
            if (i > 3) {
                flag = false;
            } else {
                System.out.println("i no es mayor de 3, debemos esperar.
" + i);
                i++;
            }
        }

        // Ejemplo básico de do while
        System.out.println("-----Ejemplo básico de do...while-----");
        i = 0;
        do {
            System.out.println(i);
            i++;
        } while(i < 5);

        // Otro ejemplo de do...while
        System.out.println("----Llamando al médico con
do...while-----");
        boolean tengoCita = false;
        do {
            System.out.println("Llamando...");
            System.out.println("Me han dado cita!");
            tengoCita = true;
        } while (!tengoCita);


        // Debemos tener cuidado con los bucles while, si hacemos mal
        // la condición de salida nos podemos quedar en un bucle
infinito

    }

}
```

Ejercicios

Ejercicio 1.
Rediseña el programa de escribir un menú y que el usuario seleccione una opción, para que si el usuario introduce un valor que no existe en el menú lo pueda intentar de nuevo.

Ejercicio 2.
Escribe un programa que imprima por pantalla un tablero de ajedrez con los caracteres X y 0, el usuario debe introducir por el teclado el tamaño del tablero. Ejemplo para tamaño 5:

```
x0x0x
0x0x0
x0x0x
0x0x0
x0x0x
```

Ejercicio 3.
Si uso un bucle puedo crear una secuencia de números:

```
1 2 3 4 5
```

Si uso un bucle dentro de otro puedo crear una tabla:

```
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45
51 52 53 54 55
```

Realiza el programa que genere esta tabla.

Ejercicio 4.
¿Tienes el programa del ejercicio exterior? Era sencillo, ¿eh?
¿Qué pasa si tengo un bucle dentro de un bucle dentro de un bucle? Tendríamos un cubo en lugar de un cuadrado.
Modifica el programa del ejercicio anterior para que genere un cubo de números. ¿Cómo te las apañaras para sacarlo por pantalla?

Ejercicio 5.
Realiza un programa que vaya pidiendo números hasta que se introduzca un número negativo y nos diga cuántos números se han introducido, la media de los impares y el mayor de los pares. El número negativo sólo se utiliza para indicar el final de la introducción de datos pero no se incluye en el cómputo.

Referencias y ampliación

Break y continue

Contenido

En la sentencia condicional switch empleabamos la instrucción break para salir del switch en el momento en el que ejecutabamos las líneas de codigo de cada caso. Esta instrucción la podemos emplear en los tres tipos de bucles (for, while y do while) para salir fuera del bucle. Su uso no es muy recomendable, ya que genera código propenso a errores y difícil de leer.

La sentencia continue, la podemos emplear para volver al inicio del bloque del bucle, sin necesidad de terminar de ejecutar la totalidad del bloque para dicha iteración.

Ten en cuenta: break sale de la iteración actual y de la totalidad del bucle, continue sale de la iteración actual del bucle, pero continua con la siguiente.

Estas dos sentencias no se usan habitualmente en los bucles, ya que reducen la legibilidad del código y se desaconseja en diferentes guías de estilo frecuentes, pero es útil que conozcas su funcionalidad.

Código de ejemplo

```
public class UT03E05BreakContinue {

   public static void main(String[] args) {
       // Ejemplo básico de break en un bucle while
       System.out.println("-----Ejemplo básico de break en
while-----");
       int i = 0;
       while(i < 5) {
           System.out.println("El valor de i es: " + i);
           break;
       }
```

```
        // Ejemplo básico de break en un bucle for
        System.out.println("-----Ejemplo básico de break en for-----");
        for(i = 0; i < 5; i++) {
            System.out.println("El valor de i es: " + i);
            break;
        }

        // Saliendo de un bucle infinito
        System.out.println("----Ejemplo saliendo de un bucle
infinito-----");
        while(true) {
            System.out.println("Oh! No! Estamos encerrados!");
            break;
        }

        // Ejemplo básico de continue
        System.out.println("-----Ejemplo continue-----");
        i = 0;
        while(i < 10) {
            i++;
            if (i == 5) {
                System.out.println("No me vas a pillar con es rima!");
                continue;
            }
            System.out.println("El valor de i es: " + i);
        }
    }

}
```

Ejercicios

Ejercicio 1.
Crear un algoritmo que cuente y muestre por pantalla el nº de dígitos de un nº introducido por pantalla, pero solamente hasta llegar a 5 dígitos, en caso de que tenga más dejaremos de contarlos

Ejercicio 2.
Rediseña el ejercicio del coste del parquímetro pero recorriendo todos los minutos que ha estado y sumando el precio de cada minuto. Emplea un tipo de bucle diferente para cada franja de

precios y usa las instrucciones break y continue para saltar bucles o iteraciones.

Referencias y ampliación

Ternario y esperar

Contenido

El operador ternario es un operador de uso poco frecuente en el lenguaje java, debido a que los desarrolladores no están muy familiarizados con su uso o que reduce la legibilidad del código.

Este operador que hace lo mismo que una sentencia if/else de carácter simple. La sintaxis de este operador es la siguiente:

```
condicion ? valor_si_true : valor_si_false
```

Donde condición es una expresión lógica que devolverá true/false, el primer valor será el retornado en caso de true y el segundo valor será el retornado en caso de false, los caracteres “?” y “:” se emplean como separadores en esta instrucción.

Ten en cuenta que el valor retornado debe recogerse en alguna variable o emplearse directamente en un contexto dado, con la siguiente estructura:

```
valor = condicion ? valor_si_true : valor_si_false;
```

Por otra parte, en esta sesión de trabajo vamos a ver dos instrucciones que nos permiten hacer esperar a nuestro programa. Estas instrucciones se emplean para generar retardos, o en programas más complejos que requieren la sincronización de diferentes procesos. En este ejemplo lo usaremos para producir retardos en la ejecución de nuestro código. Esta funcionalidad no está muy vinculada con la unidad de trabajo, pero nos va a permitir dar profundidad y versatilidad a los ejercicios y ejemplos que realicemos, además de comenzar a familiarizarnos con la gestión del tiempo en nuestros programas.

Los dos métodos que emplearemos será mediante las clases Thread y TimeUnits. Para usarlas vamos a tener que gestionar sus potenciales excepciones, este es un concepto más avanzado del curso, de momento si necesitas emplearlo copia la estructura y no es necesario que entiendas con precisión qué hacen las instrucciones try y catch.

Código de ejemplo

```
import java.util.concurrent.TimeUnit;

public class UT03E06TernarioEsperar {

   public static void main(String[] args) {
       // En este ejemplo usamos el operador ternario para hacer
       // lo mismo que podemos hacer con una estructura if/else
       System.out.println("-----Ejemplo ternario comparado con
ifelse-----");
       int x = 3, y = 7, mayor;

       if (x > y) {
           mayor = x;
       } else {
           mayor = y;
       }
       System.out.println("Con ifesle, el número mayor es: " + mayor);

       mayor = (x > y) ? x : y;
       System.out.println("Con ternario, el número mayor es: " +
mayor);

       // Incluso lo podríamos resolver en una sola línea sin la
variable mayor
       System.out.println("El mayor es " + ((x > y) ? x : y));

       // Ejemplo espera con Thread.sleep(x)
       System.out.println("-----Ejemplo de espera con Thread-----");
       System.out.println("Escóndete, te dos 2 segundos!");
       try {
           Thread.sleep(2000);
       } catch (InterruptedException e){
           // Aquí se maneja una excepción
```

```
            // esto lo vemos más adelante en el curos
        }
        System.out.println("Que voy!");

        // Ejemplo espera con TimeUnit
        System.out.println("-----Ejemplo de espera con TimeUnit-----");
        System.out.println("Escóndete, te dos 2 segundos!");
        try {
            TimeUnit.SECONDS.sleep(2);
            // Ojo! TimeUnit requiere importar la librería.
        } catch (InterruptedException e){
            // Aquí se maneja una excepción
            // esto lo vemos más adelante en el curos
        }
        System.out.println("Que voy!");
    }

}
```

Ejercicios

Ejercicio 1.
Escribe un programa que lea un número por el teclado y diga si el número es par o impar. Emplea operador ternario.

Ejercicio 2.
Escribe un programa que a partir de una variable con el número de alumnos diga si cuántos han aprobado. Para cualquier valor entero y positivo la sentencia debe hacer un uso correcto del plural/singular. Emplear operador ternario para construir dinámicamente la oración sin necesidad de almacenar en una variable.
Ejemplos de resultados:
Han aprobado 3 alumnos.
Ha aprobado 1 alumno.

Ejercicio 3.
Escribe un programa que simule la cuenta atrás del lanzamiento de un cohete. Las indicaciones por pantalla deben ser segundos reales.

Ejercicio 4.
Escribe un programa que simule una barra de carga del 0 al 100% dibujada progresivamente con saltos temporales.

Referencias y ampliación

String

Contenido

La clase String es la herramienta que empleamos en Java para trabajar con cadenas de texto, esto se debe a que no existe un tipo de dato nativo para cadena de texto en el lenguaje Java.

Cuando queremos almacenar texto en una variable podemos declarar una instancia de este objeto (más adelante en el curso se habla en profundidad de los objetos, es normal que no entiendas qué es un objeto o una instancia aun) y trabajar con ella con normalidad como lo hacíamos con el resto de variables de otros tipos, como int, double, boolean, etc.

Una de las características más interesantes de las clases son los métodos. Estos son funcionalidades que podemos usar con ello, que están ya codificadas. En este ejemplo nos vamos a centrar en ver algunos de los métodos más interesantes de la clase String, esta clase tiene más de 50 métodos, pero no tendrás que memorizarlos, simplemente entender cómo se usan algunos de los más comunes y saber cómo puedes consultar la información del resto, para cuando los necesites en el futuro.

String es una clase que se encuentra en java.lang.String y la empleamos para almacenar una secuencia de caracteres.

Código de ejemplo

```
public class UT03E07String {

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

```
        System.out.println("-----Inicializando cadenas-----");
        String cadena1 = "Hola1";
        String cadena2 = new String("Hola2");
        String cadena3 = new String(cadena2);

        System.out.println(cadena1);
        System.out.println(cadena2);
        System.out.println(cadena3);

        // Usando secuencias de escape
        System.out.println("-----Usando secuencias de escape-----");
        String cadena4 = "Tabulador \t y \nnuevas \nlineas.";
        System.out.println(cadena4);

        // Referenciando cada letra dentro de una cadena
        System.out.println("-----Referenciando cada letra dentro de la
cadena-----");
        String cadena5 = "Hola";
        System.out.println(cadena5.charAt(0) + " está en la posición
0");
        System.out.println(cadena5.charAt(1) + " está en la posición
1");
        System.out.println(cadena5.charAt(2) + " está en la posición
2");
        System.out.println(cadena5.charAt(3) + " está en la posición
3");

        // Conociendo el tamaño de una cadena
        System.out.println("-----Midiendo el tamó de una cadena-----");
        String cadena6 = "En un lugar de la Mancha de...";
        int tamanoCadena = cadena6.length();
        System.out.println(cadena6);
        System.out.println("Mi cadena tiene un tamaño de: " +
tamanoCadena);

        // Cambiando mayúsculas y minúsculas
        System.out.println("-----Mayúsculas y minúsculas-----");
        String cadena7 = "HoLa AmIgO";
        System.out.println(cadena7);
        System.out.println(cadena7.toLowerCase());
        System.out.println(cadena7.toUpperCase());

        // Reemplazando
```

```
        System.out.println("-----Reemplazando caractertes o
palabras-----");
        // Ten cuidado! existen dos métodos iguales, para caracter y
para subcadenas
        String cadena8 = ":)";
        System.out.println(cadena8);
        System.out.println(cadena8.replace(')', '('));
        String cadena9 = "¿Hola Juan, qué tal estás?";
        System.out.println(cadena9);
        System.out.println(cadena9.replace("Juan", "María"));

        // Buscando dentro de nuestras cadenas
        System.out.println("-----Buscando dentro de nuestras
cadenas-----");
        String cadena10 = "El martes entregaremos la práctica.";
        int palabraEncontrada = cadena10.indexOf("martes");
        System.out.println(cadena10);
        System.out.println("Palabra encontrada en: " +
palabraEncontrada);
            // ¡OJO! Este método tiene 4 variantes, explóralas

        // Reemplazando dentro de nuestras cadenas
        System.out.println("-----Reemplazando dentro de nuestras
cadenas-----");
        // El martes me parece pronto...
        System.out.println(cadena10.replace("martes", "viernes"));
            // Explora las variantes
            // Puedes usar el método replaceFirst si no quieres
reemplazar
            // todas las ocurrencias, solo la primera

        // Comparando cadenas
        System.out.println("-----Comparando cadenas-----");
        // ¡RECUERDA! Las cadenas no se pueden comparar con el operador
de
        // comparación ==,  este está reselvado a los tipos de datos
nativos
        // del lenguaje, para comparar cadenas debemos usar otras
formas.
        String cadena11 = "hola";
        String cadena12 = "Hola";
        // equals devuelve true o false, lo usamos en condiciones
        // de if o como condiciones de salida para búcles habitualmente
```

```
        if(cadena11.equals(cadena12)) {
            System.out.println("Las cadenas son iguales.");
        } else {
            System.out.println("Las cadenas son diferentes.");
        }
        cadena11 = "abc";
        cadena12 = "abc";
        String cadena13 = "def";
        // Con compare to tenemos un 0 si son iguales
        int compararCon = cadena11.compareTo(cadena12);
        System.out.println("abc compareTo abc resulta: " + compararCon);
        compararCon = cadena11.compareTo(cadena13);
        System.out.println("abc compareTo def resulta: " + compararCon);
        compararCon = cadena13.compareTo(cadena11);
        System.out.println("def compareTo abc resulta: " + compararCon);
            // Si son iguales retorna 0
            // Si son diferentes retorna la distancia a la cadena
alfabéticamente
        // El método compareToIgnoreCase hace lo mismo pero sin ser
sensible
        // a mayúsculas y minúsculas
        cadena11 = "abc";
        cadena12 = "ABC";
        compararCon = cadena11.compareTo(cadena12);
        System.out.println("abc compareTo ABC resulta: " + compararCon);
        compararCon = cadena11.compareToIgnoreCase(cadena12);
        System.out.println("abc compareToIgnoreCase ABC resulta: " +
compararCon);

        // Formateo de cadenas
        System.out.println("-----Formateo de cadenas-----");
        int edad = 20;
        System.out.println("Hola tengo " + edad + " años.");
        System.out.println(String.format("Hola tengo %s años.", edad));

    }

}
```

Ejercicios

Ejercicio 1.

Realiza un programa que pida por pantalla una contraseña y diga al usuario si la contraseña introducida está en un rango correcto de caracteres (de 5 a 8).

Ejercicio 2.
Realiza un programa que ordene alfabéticamente el nombre de tres alumnos, que están en tres variables String.

Ejercicio 3.
Crea un programa que permita ver pasar un # por un campo de puntos. El # debe avanzar un paso cada 100 milisegundos.
Ejemplo de algunos pasos de la ejecución:
"................................"
"#..............................."
".........#......................."
".....................#..........."
"..............................#"
"................................"

Ejercicio 4.
Crea un programa que inserte el nombre, primer apellido, segundo apellido y edad de una persona, usando el formateo de cadena. El nombre y los apellidos deben estar en variables String separadas y la edad en una variable int.
Ejemplo de ejecución:
"Pepito Rodríguez García tiene 25 años."

Ejercicio 5.
Crea un programa que limpie los espacios en blanco sobrantes de una frase. Este debe eliminar los espacios en blanco al principio, los espacios en blanco al final y todos los espacios en blanco juntos.
Ejemplo:
Cadena inicial: " Esta es mi frase favorita . "
Cadena resultado: "Esta es mi frase favorita."

Ejercicio 6.
Explora los métodos de String y utiliza alguno que no hayamos explicado en clase.

Ejercicio 7.
Existen otras clases que son similares a String, entre ellas clases "espejo" de los tipos de datos nativos del lenguaje que ya conoces. Explora los métodos de las clases Character e Integer. Dedica el tiempo que consideres necesario a estos dos últimos ejercicios, son esenciales para entender cómo funcionan las clases y cómo emplear esa funcionalidad.

Referencias y ampliación

Math

Contenido

Desarrollando nuestros programas es habitual que nos surja la necesidad de hacer cálculos matemáticos más avanzados que los que nos ofrecen los operadores + - * / %. Muchas de estas necesidades, tienen solución en la clase Math. Esta clase nos va a permitir hacer operaciones como una raíz cuadrada, una función trigonométrica, un logaritmos, buscar el máximo o el mínimo, redondeos, etc.

Pero…¡Cuidado! A diferencia de la clase String, no vamos a emplear variables del tipo Math, no usaremos la clase para invocar a sus métodos y que estos nos resuelvan ciertos problemas, pero lo haremos empleando y recibiendo las variables que ya conocemos. De esta forma: resultado = Math.método(parámetro), donde Math es fijo, método es la función que deseamos emplear y parámetro son los datos o variables que nosotros facilitamos, y en resultado almacenaremos lo que esta función nos devuelva en caso de que devuelva algo.

La clase Math se encuentra en java.lang.Math.

En este ejemplo nos familiarizaremos con algunos métodos útiles de la clase Math y su forma de uso, entre ellos uno muy interesante que nos permite generar números aleatorios.

Código de ejemplo

```
public class UT03E08Math {

    public static void main(String[] args) {

        // Raíz cuadrada de un número
        System.out.println("-----Raíz cuadrada-----");
        double resultado = Math.sqrt(800);
```

```
        System.out.println("La raíz cuadrada de 800 es: " + resultado);

        // Valores almacenados en Math
        System.out.println("-----Valores de PI y E-----");
        resultado = Math.PI;
        System.out.println("El valor de PI es: " + resultado);
        resultado = Math.E;
        System.out.println("El valor de E es: " + resultado);

        // Trigonometría
        System.out.println("-----Trigonometría-----");
        resultado = Math.sin(60);
        System.out.println("El seno de 60 es: " + resultado);
        resultado = Math.cos(60);
        System.out.println("El coseno de 60 es: " + resultado);
        resultado = Math.tan(60);
        System.out.println("La tangente de 60 es: " + resultado);

        // Máximos y mínimos
        System.out.println("-----Máximos y mínimos-----");
        resultado = Math.max(10, 8);
        System.out.println("El mayor es: " + resultado);
        resultado = Math.min(10, 8);
        System.out.println("El mayor es: " + resultado);

        // Redondeos
        System.out.println("-----Redondeos-----");
        resultado = Math.ceil(3.7);
        System.out.println("Redondeo al alza de 3,7: " + resultado);
        resultado = Math.floor(3.7);
        System.out.println("Redondeo a la baja de 3,7: " + resultado);
        resultado = Math.round(3.7);
        System.out.println("Redondeo al natural más cercano de 3,7: " +
resultado);

        // Potenciación
        System.out.println("-----Potenciación-----");
        resultado = Math.pow(5, 3);
        System.out.println("5 elevado a 3 es: "+ resultado);

        // Número absoluto
        System.out.println("-----Número absoluto-----");
        resultado = Math.abs(-14);
```

```
        System.out.println("El valor absoluto de -14 es: " + resultado);

        // Generación de números aleatorios
        System.out.println("-----Generación de números
aleatorios-----");
        double numeroAleatorio = Math.random();
        System.out.println("Número aleatorio de 0 a 1: " +
numeroAleatorio);
        // Siempre genera un número aleatorio entre 0 y 1
        for(int i = 0; i < 10; i++){
            System.out.println("Número aleatorio: " + Math.random());
        }
        // ¿Y si quiero otro rango diferente? ¡Matemáticas!
        // Los números que genera van de 0,000000... a 0,9999999...
        // Multipliucando y sumando desplazamientos podremos adaptarlos
rangos
        System.out.println("Número del 1 al 10: " + Math.random()*10+1);
        // ¡Pero lo quiero sin decimales!
        System.out.println("Número del 1 al 10 in decimales: " +
(int)(Math.random()*10+1));
        // Ahora del 80 al 100 (multiplicar y desplazar)
        System.out.println("Número del 80 al 100: " +
(int)(Math.random()*21+80));
        // rango de min a max: (int)((Math.random() * (max - min + 1) +
min)

    }

}
```

Ejercicios

Ejercicio 1.
Crea un programa que calcule la nota entera (sin decimales) que irá al boletín de calificaciones, la nota será calculada en base a las prácticas y el exámen como un float de 0 a 10, con decimales. La nota final del boletín irá de 1 a 10 y se calculará con redondeo natural, salvo en la franja de 4.0 a 4.99 que será por truncamiento.

Ejercicio 2.
Crea un programa que calcule el perímetro y el área de un círculo a partir del radio del mismo.

Ejercicio 3.
Crea un programa que resuelva el siguiente problema. Zubat, Golbat y Crobat están en una cueva, formando un triángulo. Entre Zubat y Golbat hay 25 metros, entre Golbat y Crobat hay 12 metros, el ángulo formado en la esquina de Crobat es de 20°. ¿Cuál es la distancia entre Zubat y Crobat?

Ejercicio 4.
Crea un programa que genere tiradas de un dado de seis caras. El usuario puede lanzar tres dados al mismo tiempo.

Ejercicio 5.
Crea un programa que juegue con el usuario a piedra papel tijera.

Referencias y ampliación

Funciones

Contenido

En este ejemplo vamos a entender qué son las funciones y qué rol desempeñan en los lenguajes de programación. Debido a las características de Java y su diseño vinculado a la programación orientada a objetos es posible que no entendamos todos los matices de las funciones para este lenguaje, ese entendimiento llegará cuando tratemos más en profundidad la programación orientada a objetos. En esta lección el objetivo es entender qué son las funciones desde un punto de vista generalista de cualquier lenguaje de programación.

Las funciones son fragmentos de código que permiten desempeñar una tarea muy concreta, por ejemplo, podríamos querer una función que calcule la nota media del curso en base a las notas obtenidas, una función que calcule los impuestos de una compra, una función que genere un menú, etc.

Las funciones ganan mucho valor en la reutilización de código y en mantener una base de código simple y fácil de mantener. Si descomponemos nuestro programa en fragmentos simples e implementamos estos en diferentes funciones con un propósito claro y limitado, el programa final será más claro y las funciones obtenidas podrán ser reutilizadas en distintas partes del programa o en otro programa.

Además, las funciones pueden tener parámetros de entrada y de salida opcionalmente. Por ejemplo, una función que realiza la suma de dos números se podrá llamar “suma”, esta función recibirá dos números como entrada (los dos números que queremos sumar) y nos devolverá un número con el resultado.

La declaración de esa función sería int suma(int a, int b). Esto nos indica que recibe dos int, a y b, y devuelve otro int (el que pone delante de su nombre).

No es necesario que las funciones reciban o devuelvan parámetros, es opcional. Podríamos tener la función saludo, que imprima por pantalla un saludo, pero no reciba ni devuelva parámetros, o la función esperar, que al llamarla detenga la ejecución de nuestro programa durante un segundo.

Para declarar nuestras funciones lo haremos así:

public static TIPO_RETORNADO miFuncion(TIPO_ENTRADA entrada1, TIPO_ENTRADA entrada2…) {

…

Aquí irá todo el código de nuestra función, podremos usar las variables entrada

…

return valor; Este valor debe ser de TIPO_RETORNADO

}

Si no queremos que nuestra función devuelva nada en TIPO_RETORNADO podremos void.

Para llamar a las funciones desde el código lo haremos de la siguiente forma:

```
variable = miFuncion(3, 7);
```

En caso de que no devuelva nada o no queramos guardar el valor de salida podremos usarla sin asignar:

```
miFuncion(3, 7);
```

Código de ejemplo

```
public class UT03E09Funciones {

   public static void main(String[] args) {
       // Ejemplo de código que puede emplear una función
       System.out.println("-----Ejemplo de código candidato a ser una función-----");
       double nota1 = 7.84, nota2 = 0.35, nota3 = 6.32, nota4 = 4.67;
       int notaFinal;

       // Calculamos la nota1
       if(nota1 < 1) {
           notaFinal = 1;
       } else if (nota1 > 4 && nota1 < 5) {
           notaFinal = 4;
       } else {
           notaFinal = (int)Math.round(nota1);
       }
       System.out.println("La nota final para " + nota1 + " es: " + notaFinal);

       // Calculamos la nota2
       if(nota2 < 1) {
           notaFinal = 1;
       } else if (nota2 > 4 && nota2 < 5) {
           notaFinal = 4;
       } else {
           notaFinal = (int)Math.round(nota2);
       }
       System.out.println("La nota final para " + nota2 + " es: " + notaFinal);

       // Calculamos la nota3
```

```
        if(nota3 < 1) {
            notaFinal = 1;
        } else if (nota3 > 4 && nota3 < 5) {
            notaFinal = 4;
        } else {
            notaFinal = (int)Math.round(nota3);
        }
        System.out.println("La nota final para " + nota3 + " es: " +
notaFinal);

        // Calculamos la nota4
        if(nota4 < 1) {
            notaFinal = 1;
        } else if (nota4 > 4 && nota4 < 5) {
            notaFinal = 4;
        } else {
            notaFinal = (int)Math.round(nota4);
        }
        System.out.println("La nota final para " + nota4 + " es: " +
notaFinal);

        // Alternativa usando funciones (ver funcion calcularNota abajo)
        System.out.println("-----Mismo ejemplo usando una
función-----");
        System.out.println("La nota final para " + nota1 + " es: " +
calcularNota(nota1) );
        System.out.println("La nota final para " + nota2 + " es: " +
calcularNota(nota2) );
        System.out.println("La nota final para " + nota3 + " es: " +
calcularNota(nota3) );
        System.out.println("La nota final para " + nota4 + " es: " +
calcularNota(nota4) );

        // Ejemplo de función que no devuelve ni recibe nada
        System.out.println("-----Ejemplo de función que no devuelve ni
recibe nada-----");
        saluda();

        // Ejemplo de función que no devuelve nada
        System.out.println("-----Ejemplo de función que no devuelve nada
pero si recibe-----");
        saludaA("Pepe");
```

```
        // Ejemplo de función con varios parámetros de entrada
        System.out.println("Ejemplo de función con varios parámetros de
entrada");
        int valorSuma4Numeros;
        valorSuma4Numeros = sumaCuatroNumeros(1, 2, 3, 4);
        System.out.println("La suma de los cuatro número es " +
valorSuma4Numeros);

        contadorInverso(5);


    }


    // Función para calcula la nota de boletín a partir de la nota
decimal
    public static int calcularNota(double nota) {
        int resultado;
        if(nota < 1) {
            resultado = 1;
        } else if (nota > 4 && nota < 5) {
            resultado = 4;
        } else {
            resultado = (int)Math.round(nota);
        }
        return resultado;
    }

    // Función que saluda a todo el mundo
    public static void saluda(){
        System.out.println("Hola a todos");
    }

    // Función que saluda por un nombre
    public static void saludaA(String nombre){
        System.out.println("Hola " + nombre);
    }

    // Función para sumar cuatro números
    public static int sumaCuatroNumeros(int a, int b, int c, int d){
        return a + b + c + d;
    }
```

```
    public static void contadorInverso(int cuenta) {
        if (cuenta > 1) {
            System.out.println(cuenta);
            contadorInverso(--cuenta);
        } else {
            System.out.println(cuenta);
        }
    }

}
```

Ejercicios

Ejercicio 1.
De todos los programas que has realizado en el curso hasta este momento, seguro que se te ocurre alguno que se beneficiaría de la capacidad de reutilización de las funciones. Reescribe un ejercicio anterior usando funciones.

Ejercicio 2.
Crea un programa en el que introduzcas cinco números por el teclado y diga cuál de ellos es el mayor. Toda la gestión de la obtención de los números por el teclado debe hacerse en una función con esta declaración: public static int lecturaNumeroTeclado().

Ejercicio 3.
Crea una función que lance N dados de M caras y muestre el resultado impreso por pantalla (N y M son parámetros de la función).

Ejercicio 4.
Escribe un programa que transforme números decimales a binario empleando funciones.

Ejercicio 5.
Crea un programa que use la siguiente función. Depúrala paso a paso para entender su funcionamiento.

```
public static void contadorInverso(int cuenta) {
     if (cuenta > 1) {
          System.out.println(cuenta);
          contadorInverso(--cuenta);
     } else {
          System.out.println(cuenta);
     }
```

```
}
```

Referencias y ampliación

Ámbito de variables

Contenido

Ahora que hemos comenzado a usar las funciones en Java, es importante consolidar el concepto del ámbito o ciclo de vida de las variables, esto es, entender donde nuestras variables declaradas pueden usarse y donde no, donde nacen y donde mueren.

En general podemos pensar que una variable nace y muere en el nivel de paréntesis {} en el que ha sido creada y en todos los {} que tenga por debajo.

Entendiendo esto sabremos si una de nuestras variables es visible en diferentes puntos del programa, en caso de serlo, podremos usarla, en caso de no serlo, podremos declarar una variable con el mismo nombre y usarla en ese ámbito nuevo sin que suponga un problema.

Además veremos cómo declarar variables con un valor final (constantes), que podremos usar en nuestros programas, aunque parezcan inútiles, estas variables tienen muchas ventajas respecto al uso de sus valores directamente en nuestro código.

Código de ejemplo

```
public class UT03E10AmbitoVbles {

   static int variableFueraMain = 2;
   static final int variableFinal = 10;

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

```
        // Ámbito de variables en {}
        System.out.println("-----Ámbitos de variables en diferentes
{}-----");
        int i = 1;
        int v1 = 10;

        if (i==1) {
            //int v1 = 2; //esto da error!
            int w1 = 15;
            System.out.println("El valor de w1 en el bucle es: " + w1);
        }
        //System.out.println("El valor de w1 fuera del bucle es: " +
w1); //esto da error!
        int w1 = 30;
        System.out.println("Mi nuevo w1: " + w1);

        while(i > 0) {
            //int v1 = 5; //esto da error!
            i++;
        }

        // Esta tambien da error
        //for(int v1 = 20; v1 < 18; v1--){} //esto da error!

        System.out.println("El valor de v1 es: " + v1);


        // Ámbito de las variables declaradas en for
        System.out.println("-----Ámbitos de variables declaradas en
for-----");

        for(int v2 = 1; v2 < 5; v2++){
            // esta v2 nace y muere en este bucle
        }

        for(int v2 = 20; v2 < 25; v2++){
            // esta v2 nace y muere en este bucle
            // no interfiere con la anterior
        }

        int v2 = 100;
        System.out.println("El valor de v2 es: " + v2);
```

```
        // Ámbito de variables en funciones
        System.out.println("-----Ámbito de variables en
funciones-----");
        int v3 = 5;
        funcionVariables();
        System.out.println("El valor de v3 en el main es: " + v3);

        // Variables que puedes usar dentro y fuera de las funciones
        System.out.println("-----Ámbito de variables fuera y dentro de
funciones-----");
        System.out.println("Variable fuera del main llamada desde main:
"+ variableFueraMain);
        variableFueraMain = 3;
        System.out.println("Variable fuera del main llamada desde main:
"+ variableFueraMain);
        funcionVariables2();

        // Variables final
        System.out.println("-----Variables final-----");
        System.out.println("Variable final: " + variableFinal);
        //variablefinal = 11; // esto da error!

    }

    public static void funcionVariables(){
        int v3 = 10;
        System.out.println("El valor de v3 dentro de la función es: " +
v3);
    }

    public static void funcionVariables2(){
        System.out.println("Variable fuera del main llamada desde
función: "+ variableFueraMain);
        variableFueraMain = 4;
        System.out.println("Variable fuera del main llamada desde
función: "+ variableFueraMain);
    }

}
```

Ejercicios

Sin ejercicios para este apartado.

Referencias y ampliación

Bibliotecas y paquetes

Contenido

En muchas ocasiones crearemos funciones que se pueden reutilizar en diversos programas. Cuando esto suceda, podríamos copiar nuestras funciones de un programa a otro para emplearlas, pero esto no es muy eficiente y hace que el mantenimiento del código sea tedioso y propenso a errores.

Para solucionar este problema existen las bibliotecas de funciones, o paquetes. Estas bibliotecas suelen aportar funcionalidad para resolver algún problema concreto, por ejemplo una biblioteca de funciones para realizar diferentes operaciones matemáticas, o para trabajar con cadenas de texto, o para generar diferentes tipos de laberintos.

Estas bibliotecas podrán ser exportadas e importadas por los diferentes programas para emplearlas.

Las bibliotecas de funciones, o paquetes, pueden ser creadas por nosotros mismos, creadas por un tercero, o perteneces a la lista de bibliotecas propias del lenguaje Java.

Cuando hablamos de las bibliotecas y paquetes propios del lenguaje Java, nos solemos referir a ellas como API del lenguaje (Application Programming Interface). Su organización sigue una metodología muy similar al trabajo con carpetas al que estamos acostumbrados en nuestros diferentes sistemas operativos.

Para emplear las bibliotecas de funciones o paquetes que necesitemos en nuestro programa debemos importarlas (a excepción de las que están contenidas en java.lang).

Ejemplos de importación:

```
import misfunciones.FuncionesMatematicas;

import misfunciones.*;

import java.util.Scanner;
```

Código de ejemplo

```
package misfunciones;

/**
*
* @author jar
* Esta clase ofrece funciones matemáticas simples.
*/
public class FuncionesMatematicas {
    /**
     * Esta función hace la suma de los dos números que le pases
     * como parámetros. Estos deben ser de tipo entero y el resultado
     * será un entero.
     * @param a primer número de la suma
     * @param b segundo número de la suma
     * @return devuelve la suma de los dos números
     */
    public static int sumar(int a, int b) {
        return a + b;
    }

    public static int restar(int a, int b) {
        return a - b;
    }

    public static int multiplicar(int a, int b) {
        return a * b;
    }

    public static int dividir(int a, int b) {
        return a / b;
    }
```

```
}
```

```
package ut03e11bibliotecaspaquetes;

/**
*
* @author jar
*/
public class FuncionesSaludar {
    public static void SaludoNormal(){
        System.out.println("Hola");
    }

    public static void SaludoEnIngles(){
        System.out.println("Hello");
    }

    public static void SaludoPersonalizado(String nombre){
        System.out.println("Hola " + nombre);
    }

}
```

```
import java.util.Scanner;
import java.util.StringTokenizer;
import misfunciones.FuncionesMatematicas;

public class UT03E11BibliotecasPaquetes {

    public static void main(String[] args) {
        // Usando bibliotecas y paquetes propios
        System.out.println("-----Usando mis funciones de un
paquete-----");
        System.out.println("Operaciones con 10 y 2");
        int respuesta;
        respuesta = FuncionesMatematicas.sumar(10, 2);
        System.out.println("Suma: " + respuesta);
        respuesta = FuncionesMatematicas.restar(10, 2);
        System.out.println("Resta: " + respuesta);
```

```
        respuesta = FuncionesMatematicas.multiplicar(10, 2);
        System.out.println("Multiplicación: " + respuesta);
        respuesta = FuncionesMatematicas.dividir(10, 2);
        System.out.println("División: " + respuesta);
        // Los saludos
        FuncionesSaludar.SaludoNormal();
        FuncionesSaludar.SaludoEnIngles();
        FuncionesSaludar.SaludoPersonalizado("Pepe");
        // OJO! Como la clase FuncionesSaludar se encuentra en mi mismo
        // paquete no he necesitado importarlo, pero como la clase
        // FuncionesMatematicas está en otro paquete lo he tenido que
        // importar para poder usarlo.

        // Usando bibliotecas y paquetes del lenguaje
        System.out.println("-----Usando paquetes de Java-----");
        Scanner sc = new Scanner(System.in);
        StringTokenizer st = new StringTokenizer("esto es una prueba");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }

        // Importando otros paquetes y bibliotecas
        // En el desarrollo profesional de software se emplean
        // unas herramientas conocidas como gestores de dependencias.
        // Estos gestores son unos programas que se encargan de
        // importar y mantener actualizados los paquetes que
        // empleamos en nuestra aplicación.
        // Dos de los gestores de dependencias más empleados en
        // Java son Maven y Gradle.
        // Por el momento está bien que comiences a familiarizarte
        // con estos términos, pero es pronto para comenzar a
        // trabajar con ellos.

    }

}
```

Ejercicios

Ejercicio 1.

Crea una nueva clase en el paquete mis funciones y programa alguna función para que se pueda usar desde la clase con el método main de tu programa.

Ejercicio 2.
Crea un paquete llamado FuncionesEstadisticas dentro del paquete misfunciones, crea una clase con alguna función, impórtala en tu clase con el método main y úsala.

Ejercicio 3.
Ve a la función restar de la clase FuncionesMatematicas. Borra la palabra static en la declaración de la función y familiarízate con el error que arroja nuestro programa principal. Busca información sobre esta palabra clave "static". No es imprescindible que entiendas todo lo que encuentres (más adelante en el curso hablaremos de static en clase), es solo una forma de introducirte a la búsqueda de información y resolución de problemas.

Ejercicio 4.
Explora algunas clases del paquete java.lang y de java.util en la documentación oficial de la API.

Ejercicio 5.
Busca información sobre Maven, algún vídeo simple, o algún blog servirá. Cuando lo veas/leas crea un nuevo proyecto en NetBeans, pero esta vez que sea Java with Maven -> Java Application. Echa un vistazo a los archivos que se han generado.

Referencias y ampliación

Excepciones

Contenido

Las excepciones en Java son errores que se producen mientras nuestro código se está ejecutando. Es importante entender que estos errores son diferentes a los errores de compilación, estos no nos permiten ejecutar el programa, pero los errores generados por excepciones se producen a mitad de la ejecución de nuestro programa.

En este ejemplo vamos a entender de forma básica qué es una excepción y cómo gestionarla dentro de nuestros programas, más adelante en el curso trataremos las excepciones en más profundidad.

Existen diferentes tipos de excepciones, estas excepciones intentan definir la causa del error en tiempo de ejecución, algunas son: IOException, NullPointerException, ArithmeticException o IndexOutOfBoundException. No es necesario entenderla de momento, tan solo saber que existen diferentes tipos de excepciones, y que Java, en tiempo de ejecución, puede informarnos de estas incidencias en nuestros programas.

Cuando ocurre una excepción en nuestros programas se plantean dos escenarios:

- No gestionarla. Esto provoca que la aplicación entera deja de funcionar.

- Gestionarla, informando al usuario si es necesario, pero continuando con la ejecución del programa, pese al error.

Es nuestra decisión, como diseñadores y programadores de la aplicación, decidir qué excepciones queremos tratar y cuales no, el conocimiento de las causas que las generan, la repercusión de estas en la ejecución de los programas, y la experiencia nos permitirán tomar las mejores decisiones de diseño al respecto.

Un escenario en el que es correcto tratar la excepción, puede ser un programa que deba conectarse remotamente a un recurso mediante la red, en cierta ocasiones la red no estará funcionando de forma correcta debido a problemas externos a nuestro programa. Podemos indicar este error como un fallo en la conexión de red, en lugar de interrumpir de forma abrupta nuestro programa.

Un escenario en el que es correcto no tratar la excepción, es aquel en el que un mal diseño de la codificación o el diseño de la solución está provocando los errores, por ejemplo, si estoy intentando hacer divisiones por cero en nuestro programa, este generará una excepción Aritmética, si esta división puedo evitarla con un mejor diseño del código, en lugar de gestionar la excepción puedo rediseñar mi código para que no ocurra este problema en tiempo de ejecución.

Para gestionar las excepciones usamos una estructura de código en Java conocida como try-catch-finally. Y tiene la siguiente forma:
try {

Código que potencialmente genera una excepción

} catch (Exception e) {

Código que solo se ejecuta si se produce una excepción, podemos usar información de la excepción “e”

} finally {

Bloque OPCIONAL. Código que se ejecuta siempre, tanto si se produce la excepción como si no. Suele usarse para liberar recursos.

}

Código de ejemplo

```
public class UT03E12Excepciones {


   public static void main(String[] args) {
       System.out.println("-----Usando el bloque
try-catch-finally-----");
       try {
           System.out.println("Try sin posibles excepciones.");
       } catch(Exception e) {
           System.out.println("Catch sin posibles excepciones.");
       } finally {
           System.out.println("Finally sin posibles excepciones.");
       }

       System.out.println("-----Gestionando una excepción-----");
       int n1 = 5, n2 = 0;
       try {
           System.out.println("Intentamos dividir por 0.");
           n1 = n1 / n2;
       } catch(Exception e) {
```

```
            System.out.println("Catch división por 0.");
            System.out.println("Información de e:" + e);
        } finally {
            System.out.println("Finally división por 0.");
        }

        //n1 = n1 / n2; // y si lo hago sin un bloque try-catch?

        System.out.println("-----El bloque finally es opcional-----");
        try {
            System.out.println("Intentamos dividir por 0.");
            n1 = n1 / n2;
        } catch(Exception e) {
            System.out.println("Catch división por 0.");
            System.out.println("Información de e:" + e);
        }

    }

}
```

Ejercicios

Ejercicio 1.
Realiza un programa que pida un número por el teclado, si el usuario introduce algo que no sea un número, el programa debe decirle al usuario que el dato no es válido y que introduzca un número. Usa gestión de excepciones.

Ejercicio 2.
Investiga qué implican los siguientes tipos de excepciones: IndexOutOfBoundsExpection y NullPointerException.

Referencias y ampliación

Expresiones regulares

Contenido

Las expresiones regulares son secuencias de caracteres que se emplean como patrones de búsqueda para comprobar el contenido o formato de una cadena de texto. Estas expresiones no son exclusivas de Java, se trata de una herramienta común en el desarrollo de software y en la informática en general.

Puedes aprender más sobre las expresiones regulares en los siguientes enlaces:

https://es.wikipedia.org/wiki/Expresión_regular

https://regexr.com/

https://regex101.com/

Una vez entiendas bien qué son las expresiones regulares y cómo se emplean, vamos a ver cómo usarlas en Java.

La funcionalidad para trabajar con expresiones regulares en java se encuentra en la librería java.util.regex. Esta librería incluye tres clases: Pattern, Matcher y PatternSyntaxException. Que se emplean para definir patrones, para buscar patrones y para gestionar errores de sintaxis de patrones respectivamente.

Código de ejemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class UT03E13ExpresionesRegulares {

   public static void main(String[] args) {
       // Encontrando patrones con compile sin flags
       System.out.println("-----Encontrando patrones con compile sin
flags-----");
       // así indicamos el patrón
       Pattern pattern = Pattern.compile("Zubat");
       // así buscamos el patrón en una cadena de texto
       Matcher matcher = pattern.matcher("El mejor Pokemon es Zubat");
```

```
        // así podemos saber con un booleano si el patrón ha sido
encontrado
        boolean matchFound = matcher.find();
        if(matchFound) {
          System.out.println("Patrón encontrado.");
        } else {
          System.out.println("Patrón no encontrado.");
        }
        matcher = pattern.matcher("El mejor Pokemon es zubat");
        matchFound = matcher.find();
        if(matchFound) {
          System.out.println("Patrón encontrado.");
        } else {
          System.out.println("Patrón no encontrado.");
        }

        // Encontrando partones con compile usando flags
        System.out.println("-----Encontrando partones con compile usando
flags-----");
        pattern = Pattern.compile("zubat", Pattern.CASE_INSENSITIVE);
        matcher = pattern.matcher("El mejor Pokemon es Zubat");
        matchFound = matcher.find();
        if(matchFound) {
          System.out.println("Patrón encontrado.");
        } else {
          System.out.println("Patrón no encontrado.");
        }
        // Explora el resto de flags de la clase Pattern por tu cuenta.

        // Ejemplo de validación de formato de matrícula
        System.out.println("-----Ejemplo de validación de formato de
matrícula-----");
        pattern = Pattern.compile("^[0-9]{4}[A-Z]{3}"); // suponemos que
las vocales son válidas
        matcher = pattern.matcher("2134DFG");
        matchFound = matcher.find();
        if(matchFound) {
          System.out.println("Formato de matrícula correcto.");
        } else {
          System.out.println("Formato de matrícula incorrecto.");
        }
    }
```

```
}
```

Ejercicios

Ejercicio 1.
Valida si un texto contiene la letra s en mitad de alguna palabra, no al principio o al final.

Ejercicio 2.
Valida si un texto es una dirección de correo electrónico válida.

Ejercicio 3.
Valida una dirección IPv4 en notación CIDR.

Ejercicio 4.
Valida si un texto contiene alguna etiqueta XML.

Referencias y ampliación

UT04 - Aplicación de las estructuras de almacenamiento

Arrays

Contenido

Los arrays (arreglos, vector o matriz) son variables que permiten almacenar múltiples valores de un mismo tipo.
Si necesitamos crear un programa que almacene la velocidad del viento cada minuto durante una hora, podríamos hacerlo de la siguiente forma;

```
int velVientoMin00;
```

```
int velVientoMin01;
int velVientoMin02;
int velVientoMin03;
int velVientoMin04;
[...]
int velVientoMin58;
int velVientoMin59;
```

¿Y si necesito hacerlo durante un día entero? ¿O una semana?
Los arrays permiten abordar el problema de almacenar múltiples datos de un mismo tipo de forma más sencilla.

Existen diferentes formas de declarar un array. Comencemos con el ejemplo de las muestras de la velocidad del viento.

```
int[] velViento; <- así indicamos que es un array de enteros
velViento = new int[60]; <- así indicamos el número de elementos
```

Esto se puede abreviar en una sola expresión:

```
int[] velViento = new int[60];
```

También podemos declarar e iniciar un array en una sola expresión, lo haremos de la siguiente forma:

```
int[] resultados = {2, 45, 21, 1, 27};
```

Este array será de tamaño 5 y tendrá los valores 2, 45, 21, 1 y 27 en cada una de las cinco posiciones.

Ten en cuenta que una vez defines el tamaño de un array este no podrá cambiar. Si necesitamos una estructura de datos con un tamaño dinámico no deberíamos usar un array. Más adelante, en el curso, aprenderemos a usar estructuras que permitan un número dinámico de datos.

De acuerdo, tenemos nuestro array y hemos definido el tamaño. ¿Cómo podemos asignar valores y leerlos?
Antes de asignar y leer valores es importante entender cómo funcionan los índices en los arrays. Vamos a ilustrarlo con un ejemplo:

```
inv[] myArray = new int[3];
```

Tenemos un array de tres elementos. Vamos a representarlo gráficamente, suponiendo que los valores que tiene nuestro array son 5, 10 y 15.

```
+---------+---+----+----+
| valores | 5 | 10 | 15 |
+---------+---+----+----+
```

¿Qué posición ocupa cada elemento?

```
+----------+----+----+----+
| posición | 1ª | 2ª | 3ª |
```

```
+----------+----+----+----+
| valores  |  5 | 10 | 15 |
+----------+----+----+----+
```

Y ahora lo importante, ¿qué índice accede a cada posición? El índice es el número que debemos emplear en Java para leer o escribir en la posición del array, y este siempre comienza en 0, así el primer elemento será el de índice 0, el segundo el de índice 1, el tercero el de índice 2, etc.

Con lo cual, nuestro array de tres elementos tendrá los índices 0, 1 y 2, como podemos ver en la siguiente tabla.

```
+----------+----+----+----+
| posición | 1ª | 2ª | 3ª |
+----------+----+----+----+
| índice   |  0 |  1 |  2 |
+----------+----+----+----+
| valores  |  5 | 10 | 15 |
+----------+----+----+----+
```

Ten cuidado, si un array tiene 3 elementos nunca podrá usar 3 como índice del mismo, ya que estaría fuera del rango.

Agora sí, vamos a inciar los datos y a leerlos:

```
miArray[0] = 5; <- asignación
System.out.println("Mi array 0: " + miArray[0]); <- uso
```

Para finalizar esta lección debemos saber que para conocer el tamaño de un array podemos usar la propiedad length, esta nos va a decir su tamaño. No necesita usar paréntesis al final, ya que no se trata de una función si no de una propiedad. ¡Ten cuidado! Si el tamaño de tu array es 5 los índices irán de 0 a 4.

Código de ejemplo

Ejercicios

Ejercicio 1.

Crea un programa que almacena las notas obtenidas en los 6 módulos del curso. Asigna valores a cada nota (no es necesario que sea con Scanner) y saca la nota media.

Ejercicio 2.

Crea una array de String y en cada elemento del mismo una palabra. Imprime la frase completa que forman las palabras.

Ejercicio 3.
Amplía el ejercicio 1 para tener un segundo array que contenga los nombres de los módulos. Imprime por pantalla el nombre y la nota de cada módulo, como en el siguiente ejemplo:
Programación: 7
Lenguajes de marcas: 6
etc.

Ejercicio 4.
Amplía el ejercicio 1 para tener un segundo array con las notas de otros 7 módulos del segundo curso, introduce datos para este segundo curso. Crea un nuevo array con los datos de ambos arrays, luego realiza un bucle que saque por pantalla las notas de todos los módulos del ciclo.

Ejercicio 5.
Crea un programa que le permita al usuario decir cuantos números quiere introducir, tras esto debe solicitar estos números y almacenarlos en un array. Finalmente sacará por pantalla la suma de todos los números.

Ejercicio 6.
Crea el siguiente array:
`int[] miArray = new int[];`
Explora los métodos y parámetros que te propone la javadoc al escribir:
`miArray.`

Referencias y ampliación

Arrays n-dimensionales

Contenido

En el ejemplo anterior entendíamos qué era una array y cómo lo podíamos emplear.

Los arrays tienen muchos casos de uso que favorecen su empleo, como veíamos en el ejemplo del sensor de viento, pero, ¿qué pasa si quiero almacenar los datos del viento de las 24 horas del días? Podríamos ampliar el tamaño del array a 1440 (60x24) y poner en cada posición del array el valor del viento para cada minuto del día. Otra solución es usar un array de 60 elementos y cada uno de esos arrays meterlo como dato de un array de 24 elementos, uno para cada hora. Esta solución es lo que se conoce como un array de dos dimensiones, y para inicializarlo lo haríamos de la siguiente forma:
`int[][] velViento = new int[24][60];`
Este array estará formado por 24 arrays de tamaño 60 enteros, aunque es más sencillo entenderlo como un array de dos dimensiones, en el que tenemos 24 filas con 60 columnas, para formar una matriz de 1440 elementos, con la velocidad del viento para cada uno de los minutos que tiene un día.

De la misma forma, podemos hacer arrays de tres dimensiones, cuatro, cinco, etc. Aunque podamos hacerlo tenemos que entender si los arrays de muchas dimensiones son la solución correcta para nuestro problema y si el programa resultante será fácil de leer y entender.

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una matriz bidimensional de 3x3 e inicialízala con valores enteros. Luego, calcula la suma de todos los elementos de la matriz.

Ejercicio 2.
Escribe un programa que encuentre el elemento más grande en un array tridimensional que debe ser rellenado con números aleatorios y muestre su valor junto con las coordenadas (fila y columna) donde se encuentra.

Ejercicio 3.

Escribe un programa que copie los elementos de una matriz bidimensional 3x3 a otra matriz de igual tamaño, pero volteada, y luego muestre las dos.
Ejemplo:
Normal
1 2 3
4 5 6
7 8 9
Volteada
9 8 7
6 5 4
3 2 1

Ejercicio 4.
Crea una matriz bidimensional que represente un tablero de ajedrez, donde 'B' son las casillas blancas y 'N' representan las casillas negras. Imprime el tablero en la consola. Recuerda que el tablero de ajedrez es 8x8 y que las casillas adyacentes siempre son del color contrario.

Ejercicio 5.
En un array bidimensional como este a[3][3] puedes usar a.length y a[0].length. ¿Por qué? Reflexiona sobre ellos, intenta usar a[0][0].length y prueba con arrays de más dimensiones.

Referencias y ampliación

Recorrer arrays

Contenido

En la exposición de los arrays n-dimensionales recorremos un array bidimensional mediante un doble bucle. Para recorrer este array hacemos dos índices, que son los elementos inicializados en cada bucle, ‘i’ y ‘j’, con estos índices podemos recorrer la matriz, haciendo que el índice del primer bucle representa las filas y el índice del segundo bucle representa las columnas.

Este tipo de bucles nos da un gran control sobre los elementos con los que queremos trabajar, pero requiere establecer inicialmente estos elementos y gestionarlos en cada iteración.
Cuando no necesitamos usar este índice, y tan solo queremos recorrer todos los elementos del array podemos usar una nueva versión del bucle, conocida como forEach. Con esta notación podemos recorrer todos los elementos del array sin necesidad de emplear un índice. Este tipo de bucle se emplea con objetos que emplean la interfaz iterable, esta interfaz y su uso la veremos más adelante en el curso, pero de momento usaremos esta notación con arrays para los que no necesitemos un índice cuando los queramos recorrer.

La sintaxis es:
for(tipo elemento_en_el_bucle : mi_array) {}
Donde:
“tipo” es el tipo de dato de cada elemento del array.
“elemento_en_el_bucle” es el nombre con el que voy a referenciar a cada la variable obtenida en cada interacción de array.
“mi_array” es el array que quiero recorrer.

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea un array de enteros de 100 elementos. Rellena este array con números aleatorios del 1 al 10. Crea un segundo array con la cantidad de veces que el array de elementos aleatorios tiene cada uno de los números. Imprime el resultado de este segundo array por pantalla.

Ejercicio 2.
Crea un array de Strings en el que cada elemento del array sea un String con cada uno de los enunciados de los ejercicios propuestos. Crea un segundo array bidimensional de tipo char en el que cada elemento es un carácter de cada enunciado de ejercicio.

Ejercicio 3.
Crea un array de 5x5x5x5, rellénalos de números aleatorios entre 100 y 600. Obtén el número mayor, el menor y la media.

Ejercicio 4.
Al imprimir el array en el ejemplo con la siguiente línea de código salen letras raras. Investiga qué es eso que sale.
`System.out.println("Fila: " + fila); // ???`

Ejercicio 5.
Busca qué es “Iterable” en Java. Es normal que no entiendas del todo qué es o para qué se emplea, pero intenta entender lo máximo que puedas con tus conocimientos actuales.

Referencias y ampliación

Búsqueda en arrays

Contenido

Los arrays son útiles para una gran cantidad de casos de uso en programación, uno de los casos más habituales es el de obtener información de un array de cuyos datos no tenemos una trazabilidad completa, y simplemente son un conjunto de datos comunes.

Para realizar búsquedas tenemos varias herramientas a nuestra disposición. La primera sería realizar una búsqueda con las herramientas que ya conocemos, esto es, recorrer el array con un bucle y buscar en base a comparaciones.

La otra forma es mediante la clase Arrays. Esta clase pertenece al paquete java.utils. y puedes encontrar su documentación para Java 11 en este enlace:
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html

Esta clase tiene algunos métodos (funciones) que permiten manipular y hacer operaciones habituales con arrays. En este ejemplo se ilustran las operaciones de búsqueda.

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una array con 100 elementos enteros y rellénalo con números del 1 al 100. Usando la clase Arrays encuentra si se ha generado el número 37.

Ejercicio 2.
A partir de estos dos arrays:

```
String[][] sopa = {
{"A", "E", "Y", "I", "V", "Y", "S", "A", "U", "R", "C", "W", "N", "M",
"N", "N", "I", "H"},
{"X", "H", "C", "N", "B", "X", "D", "D", "W", "M", "A", "H", "H", "D",
"U", "R", "B", "J"},
{"J", "K", "X", "I", "G", "E", "Z", "B", "Y", "K", "K", "O", "R", "E",
"V", "K", "C", "V"},
{"S", "O", "A", "V", "O", "J", "L", "U", "G", "L", "C", "O", "Q", "F",
"W", "W", "T", "Y"},
{"Q", "F", "L", "E", "L", "N", "F", "L", "D", "R", "Q", "R", "Y", "I",
"W", "R", "K", "Q"},
{"D", "F", "D", "N", "B", "H", "O", "B", "S", "C", "I", "T", "O", "I",
"G", "E", "B", "O"},
{"W", "I", "D", "U", "A", "Z", "O", "A", "M", "P", "I", "M", "B", "B",
"C", "H", "Y", "B"},
{"M", "N", "X", "S", "T", "Z", "U", "S", "Q", "J", "R", "U", "E", "A",
"A", "Q", "C", "Z"},
{"J", "G", "N", "A", "N", "O", "D", "A", "O", "F", "D", "O", "F", "R",
"U", "T", "Y", "V"},
{"X", "H", "T", "U", "G", "A", "R", "U", "J", "H", "P", "N", "U", "G",
"R", "H", "E", "C"},
{"C", "Z", "X", "R", "Q", "E", "B", "R", "Z", "U", "B", "A", "T", "T",
"R", "J", "Z", "K"},
{"J", "S", "H", "Z", "U", "Y", "U", "D", "Z", "C", "D", "W", "R", "P",
"L", "K", "Y", "B"}};
String[] palabras = {"BELLSPROUT", "BULBASAUR", "CROBAT", "GOLBAT",
"GRIMER", "IVYSAUR", "KOFFING", "MUK", "VENUSAUR", "ZUBAT"};
```

Realiza un programa que muestre la sopa de letras y la lista de palabras contenidas. El usuario debe poder seleccionar la palabra que quiere encontrar. Tras la selección se redibujará la sopa de letras con las letras de la palabra seleccionada marcada en un color diferente. Ten en cuenta que las palabras solo pueden aparecer de izquierda a derecha, de arriba a abajo y en diagonal de izquierda-arriba a derecha-abajo.

Ejercicio 3.
Realiza un programa con las siguientes líneas de código:

```
int[] a = {1,2,3,4,5};
int[] b, c;
b = a;
c = Arrays.copyOf(a, a.length);
```

Modifica algún valor del array “a” y depura para ver qué sucede con los demás arrays. Investiga lo sucedido y saca conclusiones.

Referencias y ampliación

Ordenación en arrays

Contenido

En una gran cantidad de escenarios, resolviendo problemas de programación, necesitaremos ordenar el contenido de los arrays. En ocasiones necesitaremos ordenar valores numéricos, textos u otros tipos.

Para realizar la ordenación de nuestro array podemos usar la clase Arrays, que empleamos en el ejemplo anterior, y sus métodos/funciones asociados a la ordenación. Esto se realiza con métodos sobrecargados, que permiten diferentes parámetros y diferentes tipos de datos.
Con la siguiente instrucción ordenamos el array en su totalidad en sentido ascendente.

```
Arrays.sort(miArray);
```

Con la siguiente instrucción ordenamos las posiciones de la 10 a la 20 de nuestro array de forma ascendente.

```
Arrays.sort(miArray, 10, 20);
```
Existen más herramientas para ordenar de forma descendente, o con otros patrones, que estudiaremos más adelante en el curso.

Aparte de ordenar arrays mediante la clase Arrays, vamos a ordenarlos nosotros mismos diseñando nuestros propios algoritmos. Esto nos va a permitir desarrollar más nuestra capacidad para resolver problemas, así como entender las dificultades que surgen y las posibles soluciones en este problema clásico de la algoritmia.

Para conocer algunos de estos algoritmos vamos a emplear el siguiente recursos:
https://blog.zerial.org/ficheros/Informe_Ordenamiento.pdf

Código de ejemplo

Ejercicios

Ejercicio 1.
Tenemos dos arrays. El primero tiene los nombres de los alumnos, y el segundo tiene sus notas. Puedes usar estas declaraciones en el ejercicio:
```
String[] alumnos = {"Pepito", "Juanita", "Menganito", "Fulatina"};
int[] notas = {3, 9, 7, 4};
```
Modifica el algoritmo de la burbuja para que reciba como parámetro los dos arrays, no devuela nada, pero imprima por pantalla la lista de alumnos y sus notas, ordenado de mayor nota a menor. Ten en cuenta que las notas y los nombres deben quedar emparejados igual, por ejemplo, Pepito seguirá teniendo un 3 en la lista ordenada.

Ejercicio 2.
Las siguientes muestras corresponden a datos de temperatura. Los datos que se adjuntan contienen valores atípicos. Elimina estos valores y calcula la media de temperatura, se consideran valores atípicos en esta muestra las 10 menores temperaturas y las 10 mayores temperaturas.
```
int[] temp=
{32,24,29,11,31,28,21,28,22,7,21,23,24,24,30,24,27,7,25,36,17,36,18,22,
31,27,26,28,36,36,37,28,32,27,22,27,17,26,8,29,8,23,27,31,32,23,19,37,1
8,31,36,17,29,17,25,32,9,31,29,27,37,27,18,31,20,16,9,37,29,25,29,29,16
```

```
,37,16,17,28,36,36,29,30,22,36,22,22,8,36,8,19,30,18,25,5,16,29,19,21,19,36,29,17,19,29,18,32};
```

Ejercicio 3.
Implementa las funciones que te permitan ordenar por el método de inserción y selección.

Ejercicio 4.
Implementa las funciones que te permitan ordenar por el método de shellsort.

Referencias y ampliación

UT05 - Introducción a la orientación a objetos

UT06 - Desarrollo de clases y creación de objetos

Elementos de una clase

Contenido

En la unidad anterior, entendíamos el concepto de programación orientada a objetos y el concepto de clase. Estudiamos su relevancia, motivación histórica y sus ámbitos de aplicación. Es conveniente repasar los contenidos de la unidad para afrontar esta con garantías.

En esta unidad analizaremos en detalle los instrumentos del lenguaje Java para la programación orientada a objetos y realizaremos muchos ejercicios que nos permitan dominar estos instrumentos y entender el paradigma de programación.

En esta exposición nos centramos en los elementos que conforman las clases. Las clases son el instrumento fundamental de la POO, nos permitirán definir una entidad, que relacionada con el dominio del problema que queremos resolver, servirá para contener datos y funcionalidades específicas, que podremos reutilizar en diferentes puntos de nuestro programa o en otros programas.

¿Cómo se define una clase?

La definición de una clase, normalmente, se hace en un nuevo archivo dentro de nuestro proyecto, este será un archivo con la extensión ".java", cuyo nombre deberá ser el mismo que el de nuestra clase, por ejemplo, si queremos hacer una clase llamada "Perro" deberíamos hacerlo en un archivo llamado "Perro.java".

Dentro de este archivo debemos declarar la clase de la siguiente forma:

```
public class Perro {
      // TODO EL CONTENIDO DE NUESTRA CLASE VA AQUÍ
}
```

La palabra reservada "class" indica que es una clase, y Perro será el nombre de la clase. Entre llaves tendremos todo el contenido de nuestra clase. La palabra reservada "public" permite a otras clases usar la clase Perro, pero de momento no es relevante, puesto que la estudiaremos en más profundidad más adelante.

Dentro de la clase tendremos todo el código necesario para que esta realice su cometido, el contenido serán datos (atributos) y funcionalidad (métodos).

Atributos

Los datos que necesite nuestra clase reciben el nombre de atributos. Los atributos son variables que va a tener nuestra clase de forma interna en la que almacenará valores relevantes para su correcto funcionamiento.

Los atributos pueden ser cualquier tipo de datos primitivos del lenguaje, como enteros, booleanos, caracteres o instancias de otras clases, como String, un Scanner o cualquier otra clase, tanto propia del lenguaje como creada por nosotros.

Es importante entender que en la clase declaramos las variables, pero que en cada instancia de esta (cada variable que creamos del

tipo de la clase) podrá almacenar valores diferentes dentro de cada atributo. Supongamos que creamos una clase Persona, y en esta clase tenemos un atributo llamado nombre. Si en nuestro programa necesitamos tener varias personas, el valor del atributo nombre será diferente para cada instancia. Es importante recordar que una clase es un molde, con el que crearemos objetos de ese tipo, con el molde Persona crearemos diferentes personas.

Métodos

Los métodos representan la funcionalidad de nuestras clases, van a dotar de la inteligencia y la capacidad para resolver problemas. A lo largo del curso hemos usado métodos en múltiples ocasiones aunque nos hemos referido a ellos como funciones. De ahora en adelante, y desde el punto de vista de la POO, los llamaremos métodos.

Algún ejemplo de método que hemos usado con frecuencia son los de la clase String. Observa el siguiente código

```
    String cadena = “Programación”;
    String cadena2 = cadena.toLowerCase();
```

Primero declaramos una variable de tipo String, que es lo mismo que una instancia de la clase String, y después usamos la función “toLowerCase” que es una función que no recibe parámetros y devuelve una versión en minúsculas de la cadena original. Esta función es un método de la clase String. Podríamos pensar que “toLowerCase” es un método que debería recibir como parámetro la cadena que queremos convertir, pero no es necesario, ya que al llamar a este método desde la instancia de una clase de tipo String llamada “cadena”, dentro de esta instancia existirá un atributo con el valor de ese String (no te preocupes si suena complicado haremos muchos ejemplos que ayudarán a entenderlo).

Los métodos podrán recibir y devolver parámetros tal y como lo hacíamos con lo que hasta este momento del curso llamábamos funciones, puesto que estas funciones no eran más que métodos que usábamos en nuestra clase con el main.

Y hablando del main, recuerda que en un programa solo puede existir una clase con una función main. Este será el punto por el que inicie el programa, el resto de clases aportarán funcionalidad, pero solo una clase debe tener un método main.

Constructor

El constructor es un método especial que pueden tener los objetos.

El constructor debe tener el mismo nombre que la clase y no puede retornar ningún valor (no emplea la sentencia return).

¿Qué pasa si creamos una clase sin constructor? No pasa nada, ya que el propio lenguaje crea un constructor por defecto en la compilación, pero este no tendrá ninguna funcionalidad. Cuando creamos nosotros un constructor, el compilador ya no creará este constructor vacío, y será necesario crear las instancias de la clase mediante el constructor que hemos creado nosotros.

¿Cómo uso el constructor? El constructor, como hemos indicado, es un método especial y solo se puede llamar mediante la palabra reservada “new”. Esta es una palabra que hemos usado algunas veces a lo largo del curso, por ejemplo cuando creamos instancias de la clase “Scanner”. Así que el constructor se llama siempre al crear el objeto (que es la instancia de la clase).

Palabra reservada “this”

En ocasiones, trabajando con nuestras clases podemos tener dos variables con el mismo nombre, una recibida como parámetro y otra usada como atributo de la clase, por ejemplo. Para resolver esta ambigüedad existe la palabra reservada “this”.

En la clase “Circulo”, para evitar la ambigüedad, en el constructor hemos usado el parámetro r para obtener los datos del radio. Esto ha sido una buena solución, ya que ha evitado el conflicto, pero no es la óptima puesto que “r” no nos da mucha información del parámetro, y habría sido más conveniente llamar a la variable “radio”.

```
// Esto resuelve el problema pero no es óptimo
public Circulo(double r){
      radio = r;
}
// Esto hace que nuestro código no funcione correctamente
// ya que el compilador no sabe cuando nos referimos al parámetro
// y cuando nos referimos al atributo
public Circulo(double radio){
      radio = radio;
}
// La solución óptima hace uso de “this”
public Circulo(double radio){
      this.radio = radio;
```

}

Por lo tanto, “this” hace referencia a los atributos de la propia clase.

Convenios de nomenclatura

Por último, antes de terminar esta extensa lección sobre los elementos de las clases en Java, vamos a ver un concepto importante para mejorar la calidad de nuestro código.

A lo largo del curso, hemos empleado cualquier nomenclatura que funcionase correctamente para crear proyectos, clases, funciones o variables. A partir de ahora vamos a intentar usar reglas de nomenclatura que aumenten la calidad del código.

¿A qué nos referimos con esto? Nosotros podemos nombrar las variables como queramos, veamos algunos ejemplos de declaración una variable que va a guardar la temperatura del agua:

```
int temperaturadelagua;
int TEMPERATURADELAGUA;
int temperatura_del_agua;
int TemperaturaDelAgua;
int temperaturaDelAgua;
```

Todas estas declaraciones son correctas, pero cuando trabajamos en proyectos de desarrollo de software, es habitual tomar un convenio de nomenclatura. ¿Qué es un convenio? No es más que unas normas sencillas para nombrar a las variables, las clases o los métodos de nuestros programas. Con esto conseguiremos hacer un código de mayor calidad ya que todos los desarrolladores que trabajen en el código se nombran a estas estructuras de la misma forma y el código resultante será más limpio y fácil de leer.

En Java es habitual usar un convenio de nomenclatura que emplea UpperCamelCase (también llamado PascalCase) a las clases y emplear lowerCamelCase para los métodos y variables.

Otro de los convenios habituales suele ser la selección del idioma de los métodos, variables, clases e incluso los comentarios (es habitual que sea inglés).

A partir de este punto del curso vamos a intentar trabajar con el siguiente convenio para desarrollar nuestros programas:

Para clases: UpperCamelCase
Para variables y métodos: lowerCamelCase
Idioma del código: español sin acentos ni caracteres especiales (sin ñ).
Idioma de los comentarios: español con acentos y caracteres especiales.

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una clase que permita almacenar los datos de un alumno del instituto. Crea instancias de esta clase y úsala desde otra clase con el método main.

Ejercicio 2.
Crea una clase que permita almacenar los datos de un cubo de Rubik. Crea instancias de esta clase y úsala desde otra clase con el método main.

Ejercicio 3.
Crea una clase que permita almacenar los datos de un sensor de viento (¬¬). Crea instancias de esta clase y úsala desde otra clase con el método main.

Ejercicio 4.
Crea una clase que permita hacer cálculos como una calculadora (+ - * / y raíz cuadrada), la clase no debe tener atributos solo métodos. Crea instancias de esta clase y úsala desde otra clase con el método main.

Ejercicio 5.
Modifica la clase alumno del ejercicio 1 para que tenga un constructor que reciba todos los parámetros. Crea métodos que sirvan para modificar y leer cada uno de los parámetros, y así poder hacer cualquier modificación o consulta de atributos sin emplear directamente los mismos, tan solo métodos.

Ejercicio 6.
Crea una clase llamada MovimientoPokemon. Esta clase representa los movimientos de un Pokemon (Llamarada, Paz mental, Tóxico, etc.), piensa qué atributos y métodos puede tener. Crea un constructor que permita definir sus atributos.

Ejercicio 7.
Crea una clase llamada Pokemon. ¿Qué parámetros tiene? ¿Qué métodos? Un Pokemon puede tener cuatro movimientos, ¿verdad? Haz que esos atributos sean de la clase MovimientoPokemon que has creado en el ejercicio anterior. Crea instancias de esta clase y úsala desde otra clase con el método main.

Ejercicio 8.
Modifica la clase “Circulo” para añadir un método que genere un informe por pantalla como este:

```
*****DATOS DEL CÍRCULO**************************
 - Radio del círculo: 3
 - Perímetro del círculo: 18.84955592153876
 - Área del círculo: 28.274333882308138
************************************************
```

Recuerda, los desarrolladores somos muy vagos, no hacemos las cosas dos veces si podemos evitarlo.

Ejercicio 9.
Busca información sobre UpperCamelCase, lowerCamelCase y otros convenios de nomenclatura. ¿Dependen del lenguaje? ¿o de los proyectos? ¿Qué convenios de nomenclatura suelen emplearse en Python? ¿Y para desarrollar aplicaciones Android?

Referencias y ampliación

Clases e instancias

Contenido

Tanto en la exposición de elementos de una clase, como en la unidad de trabajo anterior, veíamos la diferencia entre una clase y una instancia, pero ahora que hemos entendido qué es una clase vamos a reflexionar sobre ello y a establecer algunos términos importantes dentro de la POO.

¿Por qué es importante entender estos conceptos? Entendiendo estos conceptos no sólo puedes entender un requisito o un enunciado de un ejercicio, también entenderás la filosofía tras la programación orientada a objetos. No es necesario que memorices los términos, pero sí que tengas una reflexión clara sobre ellos para entender de qué hablamos en la POO. Tan solo entendiendo conceptualmente estos términos y sus implicaciones mejorarás tu capacidad para desarrollar.

Objeto

Programación orientada a objetos, pero ¿qué es un objeto? Un objeto es un concepto, no pienses en ningún tipo de estructura o instrucción dentro de un programa Java.

Los objetos son conceptos que trasladamos para resolver un problema. Si quiero hacer un programa que gestione la contabilidad de una empresa puedo pensar en objetos como: nómina, cuenta, impuesto, inversión. A todos esos conceptos (objetos) les atribuimos unas características y unas funciones que nos permitirán descomponer el proyecto en problemas simples y, además, podemos explicarlos a cualquier otra persona que no sepa programación.

Los objetos pueden ser cosas que podemos ver en el mundo real: coche, gato, tienda; o cosas intangibles, como: mensaje, error, resultado. Pero siempre serán cosas que estén cerca de la forma de pensar de las personas.

No podemos ver ni tocar una dirección IP, pero si estamos haciendo un programa que gestiona redes de ordenadores, el concepto IP lo entendemos como algo asociado a la problemática que queremos resolver, a esto se le llama dominio o negocio. Por ejemplo, si estamos gestionando campos de fútbol, el césped, las entradas, los jugadores, los partidos, son todo objetos relacionados con el dominio/negocio. Imagina que tu programa, además de gestionar todos estos aspectos del campo de fútbol tiene que gestionar errores, grabar datos en una base de datos, leer archivos, enviar mensajes; esto no es del dominio de los campos de fútbol, pero sí es del dominio de las aplicaciones informáticas, con lo cual se puede descomponer esa parte en objetos igualmente. La aplicación final tendrá objetos del dominio/negocio de los campos de fútbol y del dominio/negocio del desarrollo de software.

Los objetos tienen un estado y un comportamiento, por ejemplo un objeto jugador podrá ser “Pepito Pérez”, que es defensa central y puede jugar partidos de fútbol. Su estado es su nombre, su posición, su edad, etc., y su función es jugar al fútbol.

El estado de un objeto va a venir definido por sus datos, que nosotros llamaremos atributos, y su comportamiento va a venir definido por sus funciones, que nosotros llamaremos métodos.

Clase

Los objetos son algo conceptual, pero ¿cómo lo concretamos en un programa Java?

Las clases van a definir el comportamiento de los objetos desde un punto de vista genérico, son un manual de instrucciones o una plantilla que va a permitir definir cómo funciona dicho objeto.

Si tenemos una clase para los jugadores de fútbol, esta va a estar formada por sus datos (atributos) y funcionalidad (métodos), pero la clase es genérica. En nuestra clase no debe figurar la información del jugador “Pepito Pérez”, nuestra clase debe decir que un jugador debe tener un nombre, pero no establecer cual es.

Entender este concepto es muy importante. La clase solo define los atributos y métodos de un objeto, pero ella misma no se ejecuta y no hace nada, solo es una plantilla.

Instancia

Pero si la clase no se ejecuta, ¿para qué vale?

La clase vale para sacar copias de ella, y a estas copias le llamamos instancias. Si tenemos la clase jugador, y queremos que nuestro programa use dos jugadores “Pepito Pérez” y “Juanita Martínez”, lo haremos mediante instancias de la clase jugador.

Usaremos esas instancias como si fuesen variables de nuestro programa, por ejemplo podemos almacenar dichas instancia en dos variables “primerJugador” y “segundoJugador”, y de esas instancias inicializamos sus atributos y usaremos sus métodos.

A las instancias también se les llama objeto de clase, pero nosotros emplearemos el término instancia.

Interacción

¿Cómo interaccionan los objetos entre sí? Los objetos interaccionan mediante sus atributos y métodos, por ejemplo un objeto puede tener un método que retorna un valor y este valor lo podemos usar en nuestra función main o pasarlo como parámetro a otro objeto. Estas interacciones reciben el nombre de mensajes.

Ciclo de vida
¿Cuál es el ciclo de vida de una clase? ¡Ninguno! La clase es una plantilla que está ahí a la espera de ser usada, pero esta no se crea ni se destruye en ningún momento de nuestros programa, es un fragmento de código fuente que está a la espera de emplearse cuando sea necesario.

¿Cuál es el ciclo de vida de una instancia? ¡El mismo que el de cualquier variable! Todo lo que hemos estudiado sobre el ciclo de vida de las variables en el primer trimestre es aplicable a las clases.

Código de ejemplo

Ejercicios

Ejercicio 1.
Contextualiza bien las siguientes expresiones. El objetivo es entender de qué estamos hablando, queremos tener muy claros los conceptos para poder entender los enunciados de los ejercicios. Piensa a qué se refiere cada una de estas clases y pon un ejemplo similar a los que hemos realizado en clase.

- Voy a crear una clase.
- Voy a crear un objeto.
- Mi programa trabaja con la clase “Coche”, y tengo declarados 5 coches. Mi programa da un error en el método de reparación de los 5 coches. (¿Dónde tengo que arreglar el error en la clase o en la instancia? ¿Y si el error solo ocurre en un coche lo arreglo en la clase o en la instancia?)
- Tengo la clase alumno, quiero hacer una función que valga solo para los alumnos menores de edad, donde debo modificarlo en la clase o en las instancias.
- Quiero crear el alumno “Pedro”, donde debo poner este dato, en la clase o en la instancia.

Ejercicio 2.
Necesitamos un programa que maneje la información de los dos gatos de una familia, Calcetines y Bolita. Crea las clases y las instancias que consideres para gestionar los gatos. (¿Serían tus clases reutilizables para otra familia con tres gatos?)

Ejercicio 3.
Diseña (diseñar es pensar cómo será, no es necesario que lo escribas) un programa que pueda trabajar con una baraja de cartas española. Piensa qué objetos y qué instancias necesitarías. ¿Sería tu programa reutilizable para la baraja francesa?

Ejercicio 4.
Implementa el programa que has pensado en el ejercicio anterior. El programa debe poder extraer 7 cartas de la baraja. (Estas cartas no se pueden repetir, como pasaría con una baraja real)

Ejercicio 5.
Empleando la clase círculo que proporcionó el profesor en UT06E01ElementosClase, crea dos instancias de la clase con diferente valor de radio y calcula su perímetro. Establece un punto de ruptura (breakpoint) en la sentencia return del método “perimetro” de la clase círculo. Depura el programa. ¿Cuántas veces se detiene en el punto de ruptura? ¿Por qué?

Ejercicio 6.
Busca información del “garbage collector” de Java. ¿Qué es? ¿Para qué sirve? ¿Cómo se utiliza? ¿Qué papel juega en la destrucción de objetos de clase (instancias)?

Referencias y ampliación

Abstracción

Contenido

La abstracción es una propiedad muy importante de la POO. Con lo que ya hemos expuesto, podríamos entender fácilmente qué capacidades de abstracción ofrece la POO, pero vamos a reflexionar sobre ello con algunos ejemplos.

Una de las definiciones que da la real academia a la palabra abstraer es “hacer caso omiso de algo, o dejarlo de lado”, y esa es una definición muy apropiada para lo que pretendemos conseguir con la POO.

Las clases nos van a dar un patrón, una funcionalidad concreta, que cuando estemos desarrollando prestaremos atención a que se cumpla, pero cuando esté desarrollada, simplemente tendremos fe en su funcionalidad, eso mismo hemos hecho durante el curso con otras clases, por ejemplo la clase Scanner o la clase String. No tenemos ni idea de cómo es su código por dentro, pero ¡no nos importa! Lo importante es que sabemos para qué valen y cómo la podemos usar y eso es la abstracción. Alguien en su momento desarrolló esas clases para que hoy nosotros podamos usarlas sin necesidad de entender cómo están hechas, como estas clases resuelven un problema concreto, puedo basarme en la documentación que me dice cómo debo emplearlas, pero no necesito ver su código por dentro.

Lo mismo podemos hacer con las clases que nosotros creamos o las clases que crea una empresa o un proyecto. Si las clases son genéricas y resuelven un problema concreto podremos usarlas y reutilizarlas sin necesidad de supervisar o implementar de nuevo esas funciones. Imagina una empresa que hace páginas web, cada página web que hace seguramente necesita un sistema de login, pero no lo desarrolla de cero para cada página, tendrá sus clases implementadas y a la hora de usarlo en el desarrollo de una nueva página web se abstrae de ese problema, simplemente usa las clases a partir de su documentación.

También usamos la abstracción para diseñar las clases, la clase Persiana tiene que hacer esto, esto y esto, el desarrollador ya se apañará… De esta forma somos capaces de pensar la solución entera a nuestro problema, manteniendo la resolución en el dominio del problema, tal vez en el momento de codificar las clases aparezcan problemas técnicos que hagan que alguna sea inviable, en ese caso habrá que rediseñar o seguir descomponiendo el problema en otros más simples.

Código de ejemplo

Ejercicios

Ejercicio 1.
Piensa en las clases String o Scanner, ¿cómo las has empleado durante el curso?, ¿si necesitabas saber más de cómo emplearlas a donde acudías para informarte?

Ejercicio 2.
Intenta crear un programa que utilice la clase Dado facilitada por el profesor junto a esta exposición. El programa representa a dos jugadores que tiran dos dados de 20 cada uno y gana el que más puntuación obtiene de la suma de sus dos dados.

Ejercicio 3.
Analiza con detenimiento el código de la clase dado y la clase Personaje. Presta atención a los comentarios que hay sobre cada método, ¿para qué valen? Busca información sobre ello. Y otra cosa más, ¿qué atributos tiene la clase Personaje? Reflexiona sobre ello.

Ejercicio 4.
Esa clase Enemigo, no se va a quedar vacía, ¿verdad? Implementa la funcionalidad de la clase enemigo y haz un pequeño juego en el que los personajes deben luchar contra enemigos. Cuenta la cantidad de enemigos que es capaz de derrotar un personaje antes de morir.

Referencias y ampliación

Encapsulamiento

Contenido

El encapsulamiento es una de las propiedades más importantes de la POO. En la unidad de trabajo anterior y en las lecciones de esta unidad de trabajo hemos estado ejercitando el encapsulamiento, pero en esta exposición vamos a intentar definirlo de forma más precisa.

El término encapsulamiento se refiere a esa propiedad de la POO que nos permite delegar una funcionalidad a una clase, por ejemplo, si tenemos la clase “Circulo”, esta tendrá sus propios datos y su propia lógica de funcionamiento que calculará el área, el perímetro

o aquello que necesitemos. Desde otros puntos del programa no debemos preocuparnos por este funcionamiento, tan solo debemos usar dicha clase tal y como se especifica en su documentación o con la información que nos dan sus propios métodos. Si una parte del programa necesita poder calcular el área de los círculos este hará uso de la clase “Circulo” pero no le importará cómo la clase “Circulo” resuelve este problema, tan sólo entregará los datos que la clase le pida y esperará recibir el resultado que desea.

Esta propiedad, que se parece mucho a la abstracción según la definición que hemos dado hasta ahora, implica conocer cuáles son los parámetros y los métodos que la clase Círculo comparte con el resto de clases que la emplean. ¿A qué nos referimos con el término comparte? Nos referimos a que todos los métodos y atributos de la clase “Circulo” no tienen que ser usados o conocidos necesariamente por el resto de clases que se relacionen con ella. Para la clase círculo el número Math.PI es importante, pero una clase que solo necesite proporcionar un radio a un círculo para obtener un perímetro, no necesita saber del número Math.PI, ni necesita gestionar dicho número dentro de la clase “Circulo”.

Pensemos otro ejemplo, en un programa para gestionar la venta de billetes de autobús. La edad del cliente, los impuestos del momento, o algunas bonificaciones especiales pueden afectar al precio final del billete, pero la clase encargada de seleccionar el asiento del pasajero tal vez no necesite esta información, o la propia clase del pasajero, no tiene que conocer los descuentos que se aplican a mayores de edad, pero sí conocer la fecha de nacimiento del pasajero. En muchas ocasiones la totalidad de la clase y su problemática asociada no es del interés de las demás clases, el encapsulamiento pretende agrupar los datos y métodos que el objeto necesita y ofrecer funcionalidad a las demás clases.

Uno de los ejemplos más claros del encapsulamiento es la gestión de los datos de la clase. Hasta ahora hemos inicializado los datos de la clase mediante el constructor o accediendo directamente a ellos para asignarles valores. En muchas ocasiones el estado interno de la clase impone restricciones sobre su propio uso. Por ejemplo, una clase “JugadorDeFutbol” podrá jugar partidos de fútbol, o podrá ser convocado por su selección, pero si el jugador se encuentra lesionado no podrá hacerlo. En este escenario, la clase “SeleccionDeFutbol” podrá intentar usar el método “convocar” de un jugador, que normalmente respondería atendiendo a la convocatoria, pero en un caso concreto, el jugador, al estar lesionado, arrojará

una respuesta negativa a la petición de “convocar”, esto depende del estado interno de la clase “JugadorDeFutbol”. En las próximas exposiciones y los próximos ejercicios verás múltiples ejemplos que te permitirán entender bien el concepto.

Normalmente, en la POO, no se usan los atributos de una clase de forma directa (por implicaciones en la seguridad e integridad de la clase, que entenderás en próximas exposiciones), en lugar de esto, para acceder a los datos de los atributos se emplean unos métodos especiales llamados getters y setters (del inglés get y set), estos métodos permiten asignar un valor o leer el valor de un atributo. Como en el siguiente ejemplo:

```
// atributo de la clase
int a;

// método set establece su valor
public void setA(int a) {
      this.a = a;
}

// método get obtiene su valor
public int getA() {
      return a;
}
```

En próximas exposiciones entenderemos algunas funciones adicionales del encapsulamiento.

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea la clase “Jugador” para un equipo de baloncesto. Esta clase debe tener el nombre, la fecha de nacimiento, altura, el número de dorsal y la posición (las posiciones posibles deben ser: base, escolta, alero, ala-pivot y pivot).

Ejercicio 2.
Crea métodos getters y setters manualmente para los atributos de la clase “Jugador” del ejercicio anterior. Investiga si existe una forma alternativa de crearlos usando NetBeans.

Ejercicio 3.
Modifica el método set creado en el ejercicio 2 para impedir que se pueda poner una fecha de nacimiento futura al momento actual.

Ejercicio 4.
Crea la clase “Equipo” para un equipo de baloncesto. Esta clase debe tener un nombre, una ciudad y cinco jugadores. Los jugadores deben ser objetos del tipo de la clase “Jugador”.

Ejercicio 5.
Establece métodos getter y setter para la clase, además métodos para añadir y quitar jugadores. Si ya hay cinco jugadores debe dar un error al intentar añadir nuevos jugadores.

Ejercicio 6.
Crea un método en la clase “Equipo” que permita conocer la franja de edad en la que se encuentran los jugadores. El resultado debe ser: sub-16, sub-18, sub-21 o absoluta.

Referencias y ampliación

Modificadores de acceso

Contenido

Los modificadores de acceso son palabras reservadas del lenguaje que nos van a permitir controlar quién puede acceder a las clases, sus atributos y sus métodos.

¿Por qué necesitamos controlar el acceso a estos elementos? Debido a la abstracción y el encapsulamiento. En las secciones anteriores hemos entendido la utilidad de delegar el funcionamiento completo a una clase, no necesitamos entender qué hace por dentro.

Imaginemos el funcionamiento de un coche en el mundo real. Yo como usuario de un coche quiero poder abrir y cerrar las puertas,

conducirlo, rellenarlo con combustible, y funciones similares. El coche es una máquina muy compleja de la cual no necesito entender totalmente, ¿qué es un alternador? como usuario no necesito saberlo, ni entender para qué vale, solo necesito que funcione correctamente para que el coche pueda conducirse. Es por ello que nosotros no tenemos un botón junto al volante que sirva para comprobar el estado del alternador o para iniciar su carga. No necesito entender cómo funcionan los sensores del depósito de combustible, solo necesito ver el indicador de la cantidad de este que está disponible, no necesito entender cómo funciona la hidráulica asociada a la dirección asistida, solo entender que si giro el volante girarán las ruedas.

Si yo pudiese controlar la carga y descarga del alternador en mi rol de conductor de un vehículo, lo más probable es que terminase por estropearlo o no fuese capaz de conducirlo. Como conductor yo lo que necesito es que el alternador funcione correctamente y haga sus funciones cuando meto la llave en mi coche y quiero arrancarlo.

Los modificadores de acceso son la herramienta que nos va a permitir dar a los conductores las herramientas que necesitan para conducir el coche, teniendo en cuenta que ellos no necesitan entender el funcionamiento interno del mismo. El método arrancar nos permite arrancar el coche, y será la clase por dentro la que gestione si el alternador está en el estado correcto para poder hacerlo.

Lo modificadores de acceso que podemos emplear con nuestras clases, atributos y métodos son los siguientes:

Ámbito / Modificador	en la clase	en el paquete	en una subclase	fuera del paquete
private	**SI**	**NO**	**NO**	**NO**
protected	**SI**	**SI**	**SI**	**NO**
public	**SI**	**SI**	**SI**	**SI**
sin modificador	**SI**	**SI**	**NO**	**NO**

Estos modificadores se anteponen a las clases, métodos y atributos, como en los siguientes ejemplos:

Así, cuando algo sea “public” podrá usarse en cualquier punto del programa, y cuando algo sea “private” solo podrá usarse dentro de la clase.

Para nosotros, como desarrolladores, lo más sencillo sería hacer todo “public”, así podré usar todo en cualquier parte sin necesidad de tener que pensar qué modificador de acceso uso con cada clase, atributo o método. Esto es cierto, pero genera un código de pésima calidad.

¿Qué directriz debería seguir? Deberías dar a cada clase, atributo o método los permisos más restrictivos posibles, pensando que de base todo debería ser “private”, y solo dando permisos menos restrictivos en caso de que realmente se necesite. Es más, incluso en caso de que un atributo deba usarse fuera de la clase, lo más conveniente es que siga teniendo un modificador de acceso de tipo “private” y que podamos acceder a esa información mediante un método “get”.

Código de ejemplo

Ejercicios

Ejercicio 1.
Piensa en ejemplos similares a los del coche. Diseña una clase y piensa qué modificadores de acceso debe tener cada atributo y método.

Ejercicio 2.
¿Recuerdas los ejercicios que hiciste en la primera sección de esta unidad de trabajo? Revisalos y cambia los modificadores de acceso para seguir la directriz de darle a todos los métodos y atributos los mínimos permisos necesarios.

Ejercicio 3.
En la tabla de los modificadores de acceso hay una sección que nos indica si se permite el acceso a una subclase, pero ¿qué es una subclase? En la siguiente unidad de trabajo aprenderemos qué son las

subclases, pero ahora investigalo y busca algún ejemplo que justifique el uso de subclases.

Referencias y ampliación

Otros modificadores

Contenido

En la lección anterior aprendimos el uso de los modificadores de acceso (public, private y protected), estas palabras reservadas del lenguaje se anteponen a nuestras clases, atributos o métodos para indicar desde donde podían ser accedidos.

En esta lección veremos algunos modificadores más que podemos usar para determinar más características de nuestras clases, atributos y métodos. A continuación se listan todos los modificadores que no hemos visto en lecciones anteriores, pero algunos de ellos tendrán sentido para nosotros cuando veamos futuras unidades de trabajo, o en módulos del segundo curso, los modificadores que son relevantes para esta unidad de trabajo aparecen en negrita y tienen una explicación más detallada en esta lección.

Modificadores de clase:

- final: Este modificador indica que ninguna clase nueva podrá heredar de nuestra clase.
- abstract: Este modificador indica que no se pueden crear instancias de esta clase. Para que el código de esa clase pueda ser usado de forma efectiva otra clase tendrá que heredarlo.

Modificadores de atributos y métodos:

- **final**: Los atributos y métodos con este modificador no pueden ser sobreescritos o modificados.
- **static**: Los atributos y métodos con este modificador pertenecen a la clase, no a las instancias de la misma.
- abstract: Los métodos con este modificador deben pertenecer a una clase abstracta (con este mismo modificador). Los métodos con este modificador no deben tener un cuerpo, es decir no

tiene contenido entre llaves, y este contenido debe ser proporcionado por las clases que hereden de mi clase.
- transient: Los atributos y métodos con este modificador son transitorios y no deben ser serializados cuando se serializa la clase.
- synchronized: Los métodos con este modificador sólo podrán ser accedidos por un hilo al mismo tiempo.
- volatile: Los atributos con este modificador no pueden guardar su valor en la caché y lo deben hacer siempre en la memoria principal.

Atributos final

Estos atributos ya los hemos empleado en unidades de trabajo anteriores. Se emplea cuando no queremos que el valor de una atributo pueda ser modificado, este tendrá sentido para valores que debemos emplear en varios puntos de nuestra clase y que su modificación no tiene sentido en tiempo de ejecución.

Por ejemplo, si tenemos una clase que calcula impuestos y debe usar el IVA en múltiples operaciones podríamos crear una variable que almacene el valor del IVA.

```
final int VALOR_IVA = 21;
```

En todas las operaciones de nuestro código haremos los cálculos con VALOR_IVA en lugar de con el número 21. No hay ninguna diferencia en usar VALOR_IVA o 21, pero tiene una ventaja. Imagina que VALOR_IVA se emplea en 80 lugares diferentes de nuestro código, en un momento dado, un cambio legislativo hace que el nuevo IVA sea del 18%. Para actualizar nuestro programa tan solo debemos cambiar el valor de la variable a 18, y en lugar de esta variable hemos puesto el valor 21 en cada operación tendremos que cambiar manualmente todos estos valores en nuestro código, esto genera un código menos mantenible y menos legible.

En la siguiente unidad de trabajo entenderemos el uso del modificador final para métodos.

Atributos y métodos static

En unidades de trabajo anteriores usábamos los atributos y métodos estáticos sin entender su uso, sobre todo lo empleamos para crear funciones, lo que ahora llamamos métodos.

Esencialmente, este modificador nos indica que podemos usar los atributos y métodos que lo posean de forma estática directamente sobre la clase, sin necesidad de generar una instancia de la misma.

Si tenemos la clase Saludar y esta clase tiene los métodos “darLosBuenosDias” y “darLasBuenasNoches”, si estos métodos no son estáticos deberíamos usarlos de la siguiente forma:

```
Saludar s = new Saludar();
s.darLosBuenosDias();
s.darLasBuenasNoches();
```

Pero, ¿qué nos aporta “s”? ¿Qué diferencia tendría esa instancia con una segunda instancia “s2”? Ninguna, ya que en nuestra clase supuesta no tenemos atributos o estado interno que influencian el comportamiento de la clase. Si los métodos fuesen estáticos podríamos usarlos de la siguiente forma:

```
Saludar.darLosBuenosDias();
Saludar.darLasBuenasNoches();
```

Esto sería código ejecutado directamente de la clase, no de la instancia. Cuando usábamos la clase Arrays, estábamos usando sus métodos estáticos.

Este mismo comportamiento podemos usarlo con atributos, es más cuando usamos el número pi, lo estamos haciendo de forma estática.

Math.PI (Math es la clase) (PI es un atributo estático)

Código de ejemplo

Ejercicios

Ejercicio 1.
Explora la clase Arrays (en NetBean Ctrl+Click Izquierdo o botón derecho del ratón -> Navigate -> Go to source). Presta atención al uso que hace de los modificadores, ¿hay algo que no entiendes?

Ejercicio 2.
En este momento ya tienes un conocimiento bastante extenso de la POO y sus principios de funcionamiento. Vamos a plantear un ejercicio completo en el que debes usar tus conocimientos adquiridos.
Vamos a crear un programa que permita hacer operaciones con puntos, rectas y círculos en un plano cartesiano (ejes x e y). Crea las siguientes clases:

- Punto (formado por las coordenadas x e y)
- Recta (formada por dos puntos)
- Circulo (formada por un punto y un valor de radio)
- PlanoCartesiano (una clase que permite trabajar con puntos, rectas y círculos) el plano debe poder generar los tres elementos calcular la distancia entre dos puntos, calcular coordenadas de intersección de dos rectas y calcular puntos de corte de rectas y círculos.

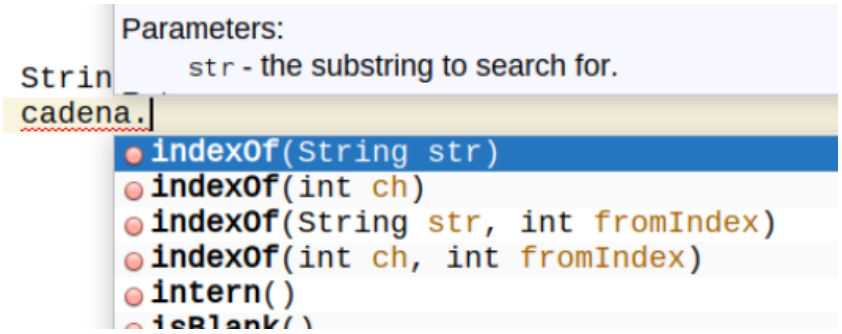
Plantea una solución evaluando todos los principios de funcionamiento que hemos trabajado en clase (clases, instancias, abstracción, encapsulamiento, modificadores) y después desarrolla tu solución. No tengas miedo, no son matemáticas avanzadas, son problemas sencillos para los que estás perfectamente preparado.

Referencias y ampliación

Sobrecarga

Contenido

En la siguiente captura estamos viendo algunos métodos que nos ofrece “cadena” que es una instancia de String. El método “indexOf” aparece varias veces, ¿es esto un error? ¿cual debo usar? ¿en qué se diferencian?



La existencia de varios métodos con el mismo nombre no es un error, es una técnica de la programación orientada a objetos llamada sobrecarga. La sobrecarga consiste en crear métodos con el mismo nombre pero que reciben diferentes parámetros, y que pueden devolver diferentes tipos. Si nos fijamos en los “indexOf” de la imagen, vemos que todos se diferencian en los parámetros que reciben, esto nos va a permitir hacer la misma operación pero con diferentes condiciones de entrada.

Un ejemplo más sencillo puede ser la sobrecarga de un método que sume:

```
public int sumar(int numero1, int numero2)
public float sumar(float numero1, float numero2)
public double sumar(double numero1, float numero2)
public double sumar(double numero1, double numero2)
public int[] sumar(int[] numero1, int[] numero2)
```

En estas cabeceras de métodos podemos ver que todos tienen el mismo nombre, pero difieren en parámetros de entrada. Y si tengo varios, ¿cómo sabe Java cuál debe emplear? Fácil, Java emplea el que se ajuste a los tipos de datos que le estás proporcionando.

Este concepto de la sobrecarga se puede aplicar a los constructores de la misma forma que a los métodos. En muchas ocasiones necesitaremos poder construir objetos de diferentes formas.

Una peculiaridad de la sobrecarga de constructores es la instrucción “this()”. Esta instrucción especial se puede emplear como primera línea de los constructores sobrecargados, y permitirá, dentro de un constructor, llamar a un constructor que sea compatible con los parámetros que le pasamos y luego seguir ejecutando nuevas instrucciones en el constructor sobrecargado.

```
public Coche(String matricula){
    this.matricula = matricula;
}

public Coche(String matricula, String modelo){
    this(matricula);
    this.modelo = modelo;
}
```

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una clase círculo, que esté compuesta por un punto y el radio. La clase debe poder construirse sin datos (crea un círculo de 1 de radio en el punto 0,0), solo con radio (en 0,0), solo con punto (radio 1).

Ejercicio 2.
En la clase BajaraSobrecargada que viene con el ejemplo de código de esta sección. Crea métodos sobrecargados para repartir cartas, estos pueden ser: vacío, con número de cartas a repartir, con número de cartas a repartir y número de jugadores.

Referencias y ampliación

Clases anidadas y enumerados

Contenido

En esta lección vamos a estudiar dos tipos particulares de clase, que son las clases anidadas y los enumerados.

Clases anidadas
Las clases anidadas son clases que podemos crear dentro de otra clase (dentro de las llaves con el contenido de la clase). ¿Para qué sirve una clase anidada? Las clases anidadas no aportan una funcionalidad nueva que no podamos desarrollar sin anidar, pero ayudan a favorecer la filosofía de encapsulamiento. Si para desarrollar la funcionalidad de una clase consideramos conveniente tener otra clase pero no aporta valor como clase autónoma, podemos definirla como clase interna.

Por ejemplo, en un programa de simulación de coches, podemos tener la clase motor, esta clase aportará funcionalidad para que se use dentro del programa, pero desarrollando podemos querer dividir su funcionalidad en otras clases que nos ayuden en su codificación y hemos decidido crear una clase anidada bujía. Si necesitase usar la bujía en varios puntos de mi programa no tendría sentido que esta fuese una clase interna de la clase motor, pero si solo se usa dentro del motor, puedo querer desarrollarla como clase interna.

Aunque las clases internas pueden favorecer el encapsulamiento, la decisión de usarlas dependerá de las preferencias del desarrollador o del equipo de desarrollo. Es posible que un equipo de desarrollo decida no usar clases anidadas como parte de su guía de estilo, si el equipo de desarrollo estima que esto es positivo en algún aspecto como la mantenibilidad del código, el diseño de pruebas, etc.

En el siguiente ejemplo vemos cómo declarar una clase anidada.

```
public class Coche {

      String marca;

      public Coche(String marca){
          this.marca = marca;
      }

      public void arrancarChoche(){
          Bujia b = new Bujia();
          b.hacerCosasDeBujias();
          System.out.println("¡El coche arranca!");
      }

      class Bujia { // clase anidada
          public void hacerCosasDeBujias(){
              System.out.println("La bujía está haciendo su
función...");
          }
      }
}
```

Las clases anidadas pueden tener modificadores de acceso private, protected o vacío. Lo más habitual es que sean private, ya que es la mejor solución para favorecer el encapsulamiento.

Las clases anidadas pueden tener el modificador static, y ser usadas de forma estática en su clase externa, sin necesidad de crear una instancia.

Una característica importante de las clases anidadas es que pueden usar los atributos de la clase externa.

Enumerados

Los enumerados son un tipo especial de clases. Estas clases permiten limitar la cantidad de valores que puede tener una variable a un rango acotado, como podemos hacer con un boolean, pero en este caso con una cantidad de valores concretos y cada uno de estos establecidos por el propio usuario.

Si queremos simular el comportamiento del lanzamiento de una moneda, podemos usar una variable boolean y decir que true es cara y false es cruz. Pero, ¿y si quiero hacer una variable con los valores de piedra, papel o tijera?

Así podemos hacerlo:

```
public enum Piepaptij {
      PIEDRA, PAPEL, TIJERA;
}
```

En NetBeans y otros IDE podemos crear un nuevo enumerado de forma similar a como añadimos una nueva clase.

Como ves en el ejemplo, los convenios de nomenclatura habitualmente sugieren que los valores de los enumerados sean palabras con todas las letras en mayúscula (y guión bajo si es multipalabra), pero Java no dará error si lo hacemos de otra forma.

Podemos iterar los valores de un enum en un bucle for each en caso de necesitarlo.

Podemos crear parámetros y constructores para nuestros enumerados, y así asociar automáticamente valores en su construcción a los estados, en este ejemplo se asocia en un enumerado con los días de la semana un valor numérico para cada día.

```
public enum DiasSemana {
      LUNES(1),
      MARTES(2),
      MIERCOLES(3),
      JUEVES(4),
      VIERNES(5),
```

```
        SABADO(6),
        DOMINGO(7);

        final int valorDia;

        private DiasSemana(int valorDia) {
            this.valorDia = valorDia;
        }
}
```

Código de ejemplo

Ejercicios

Ejercicio 1.
Rediseña las clases Jugadores y Equipo de la lección Encapsulamiento para que sean todas clases anidadas de una de ellas.

Ejercicio 2.
En el ejemplo proporcionado del enumerado DiasSemana el atributo valorDia es final. ¿Es correcto? ¿Es un error? Intenta justificar las respuestas.

Ejercicio 3.
Rediseña la clase Personaje usada en la lección sobre Abstracción para que los tipos de personaje sean enumerados.

Ejercicio 4.
Intenta rediseñar la clase baraja con la que hemos trabajado en esta unidad de trabajo (cualquier versión de ellas) para que use tanto enumerados como clases anidadas.

Paso por valor y por referencia

Contenido

En esta lección vamos a entender dos conceptos muy importantes en programación, que son el paso por valor y el paso por referencia.

Estos conceptos no son exclusivos de la programación orientada a objetos, y cada lenguaje de programación puede tener algunas características específicas al respecto. El objetivo de la lección es entenderlo conceptualmente desde un punto de vista de lógica de la programación y ver cómo se trabaja con este concepto en Java.

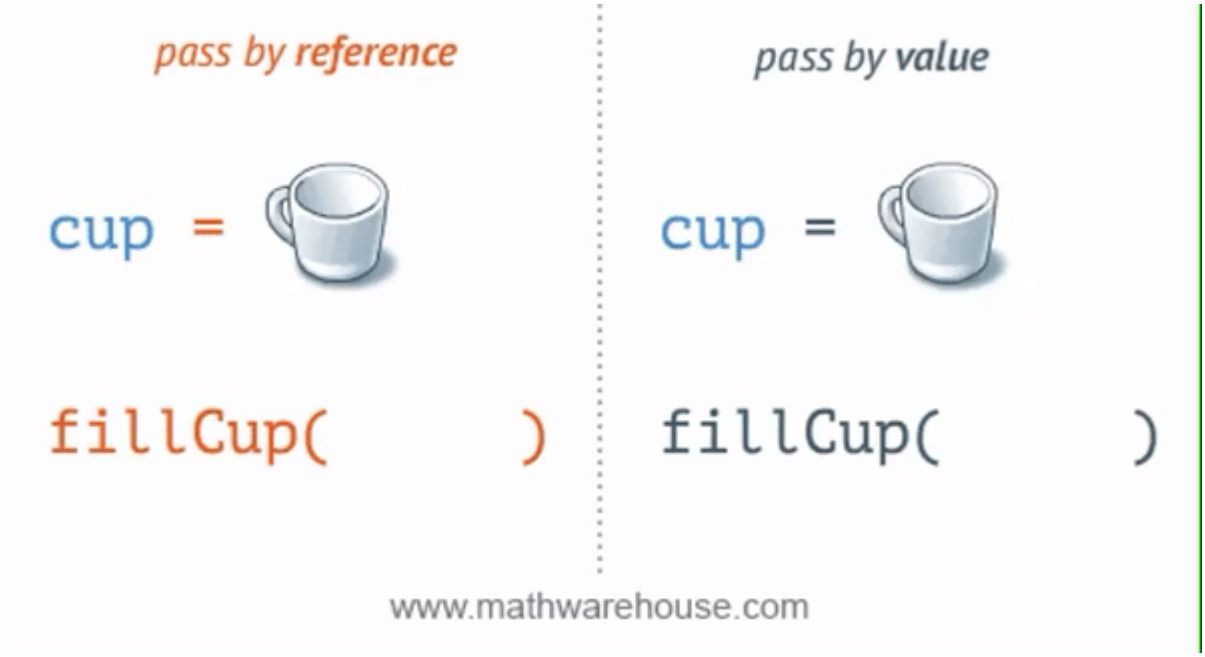
Desde el punto de vista de las ciencias de la computación, podemos pasar a una función un dato por valor, o por referencia. ¿Qué significa esto?

Si pasamos el dato por valor
Nuestro programa tiene una variable x = 5 en la posición de memoria 100, pasando por valor se hará una nueva copia en memoria de esa variable con el mismo valor, ahora tenemos en la posición 100 x = 5 y en la posición 200 x = 5, esta segunda variable es la que le estamos pasando a nuestra función, con lo cual si la función modifica la variable y la pone a 6, en nuestra memoria tendremos:

- en la posición 100: x = 5
- en la posición 200: x = 6

Ya que la función ha trabajado con la posición de memoria 200, por ello el contenido de la posición de memoria 100 no se ha visto alterado.

Si pasamos el dato por referencia
Nuestro programa tiene una variable x = 5 en la posición de memoria 100, pasando por referencia, le estamos diciendo a la función que el dato que necesita se encuentra en la posición de memoria 100. La función hará sus modificaciones sobre esta posición de memoria, con lo cual ahora el valor de x pasará a ser 6, tanto nuestra función como el programa desde el que hemos llamado a la función consideran que x está en la posición 100, con lo cual ahora las dos partes del código entienden que el valor de x es 6.



¡Recuerda! Esto es un concepto genérico de ciencias de la computación, cada lenguaje de programación hará diferentes gestiones de esta casuística, algunos lenguajes no permitirán uno de los dos métodos y otros tendrán sus propias normas o formas de trabajar con los dos métodos.

¿Cómo funciona el paso por valor y el paso por referencia en Java?
Como norma general, en Java:

- los tipos primitivos (int, boolean, float, etc.) se pasan por valor.
- los objetos se pasan por referencia.

Es importante entender que esto ocurre al copiar una variable en otra. Si copiamos un objeto en otro estaremos copiando la referencia del uno en el otro, si copiamos un tipo primitivo en otro estamos copiando su valor.

Existen algunas excepciones a esta norma general, por ejemplo, la clase String, podemos hacer copias de variables de tipo String igualándolas y al cambiar el valor de una de ellas la otra se mantendrá igual. Esto se debe a que la clase String es una clase inmutable.

¿Qué son las clases inmutables? Son clases cuyo valor se mantiene constante una vez han sido creadas y no hay ninguna forma de modificarlas.

¿Cómo podemos crear clases inmutables? Para que una clase sea inmutable debe cumplir varios requisitos:

1. La clase debe tener el modificador final (por cuestiones de herencia que veremos en la siguiente unidad de trabajo).
2. Todos los atributos de la clase deben ser private para no permitir el acceso directo a ellos.
3. No debe tener métodos de tipo “setter”.
4. Todos los atributos que puedan adquirir un valor deben se final, para que este valor solo pueda ser asignado una vez.
5. Todos los valores que necesitemos inicializar lo haremos en el constructor.
6. Si hacemos algún método que cree copias de nuestros objetos inmutables este debe hacerlo mediante una generación de nueva instancia.

Si una clase cumple todas estas características podríamos considerarla inmutable, esto en ocasiones hace que trabajar con ella sea más sencillo y evite posibles errores de cambio de estado. En cualquier caso, lo más importante es entender cuando se pasan valores y cuándo referencias y saber programar en torno a este conocimiento.

Código de ejemplo

Ejercicios

Ejercicio 1.
Estudia y entiende el ejemplo de código proporcionado con la lección.

Ejercicio 2.
Crea una clase círculo que tenga un radio y un punto (el punto debe ser de la clase punto con coordenadas x e y, como en otros ejemplos de esta unidad de trabajo). Asegúrate que la clase es inmutable cumpliendo todas las condiciones de inmutabilidad. Debe tener un método que genere una copia del punto que representa su centro.

Bonus: ArrayList para clases

Contenido

En esta sección vamos a estudiar una estructura de datos llamada ArrayList. El estudio de esta estructura y de otras similares lo veremos en profundidad en la unidad de trabajo “Colecciones de datos” sobre la que trabajaremos más adelante en el curso. La inclusión de esta estructura de datos en este punto del curso se debe a la versatilidad que nos ofrece ArrayList a la hora de trabajar con objetos. Por ello, en esta unidad veremos ArrayList desde un punto de vista del trabajo con los objetos pero sin profundizar en otras características de las colecciones de datos que veremos más adelante en el curso.

¿Para qué vale esta herramienta? ¿Qué me aporta frente a otras que ya conozco?
Solo por su nombre “ArrayList” ya podemos intuir que se trata de un Array, y así es, pero tiene la característica de que su tamaño es dinámico, lo que nos permitirá añadir y quitar elementos durante la ejecución del programa.

ArrayList se encuentra en java.util, con lo cual necesitaremos importarlo en nuestros programas cuando queramos hacer uso de esta herramienta.

```
import java.util.ArrayList;
```

Otra de las características principales de ArrayList es que solo puede contener objetos, no tipos primitivos del lenguaje como int, float, boolean, etc. Pero podemos usar las clases wrapper de los tipos primitivos del lenguaje, no podremos tener un ArrayList de int, pero si de Integer.

Así podemos declarar un ArrayList:

```
ArrayList<String> cosas = new ArrayList<String>();
```

String, como hemos visto durante el curso, es una clase, con lo cual podemos usarlo para hacer un ArrayList.

```
ArrayList<int> enteros = new ArrayList<int>(); // Incorrecto

ArrayList<Integer> enteros = new ArrayList<Integer>(); // Correcto
```

Algunos de los métodos más comunes de ArrayList son los siguientes:

- add: para añadir elementos
- get: para obtener un elemento
- set: para sobreescribir un elemento
- remove: para borrar un elemento
- clear: para borrar todos los elementos
- size: para saber el número de elementos
- contains: para saber si un elemento está en nuestra lista
- indexOf: para saber la posición de un elemento
- isEmpty: para saber si la lista está vacía
- toArray: para convertir nuestro ArrayList en un Array.

Código de ejemplo

Ejercicios

Ejercicio 1.
Investiga en la javadoc, qué métodos y características interesantes tiene ArrayList. Usa los siguientes métodos: clone, toArray, contains, isEmpty.

Ejercicio 2.
Usando ArrayList de la clase Carta proporcionada con el ejemplo, crea un programa que tenga un mazo de cartas, la mano de dos jugadores y una pila de descartes. Crea métodos que permitan mover cartas entre el mazo, las dos manos de los jugadores y una pila de descartes. El programa debe poder mostrar el contenido de cada elemento.

Ejercicio 3.

Ha llegado la hora. Hemos crecido, hemos madurado, nos hemos enfrentado a temibles enemigos, y ahora que somos más fuertes debemos enfrentarnos al mayor de nuestros miedos. Rediseña el ejercicio del sensor de viento usando las nuevas herramientas que has aprendido en esta unidad, clases, enumerados, ArrayList, etc. En lugar de tener unas muestras fijas el sensor debe generar aleatoriamente entre 50 y 100 muestras al inicio del programa.

Referencias y ampliación

UT07 - Utilización avanzada de clases

Herencia

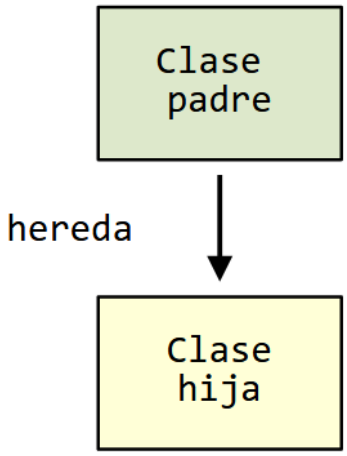
Contenido

En la unidad de trabajo anterior trabajamos en profundidad con el paradigma de programación orientado a objetos, entendíamos de forma conceptual y procedimental cómo diseñar programas con este paradigma y las características del paradigma desde un punto de vista de las ciencias de la computación y del lenguaje Java.

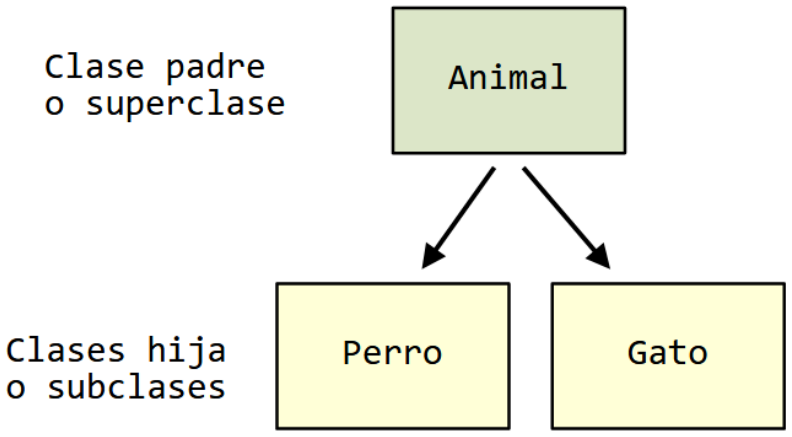
En esta unidad de trabajo vamos a entender algunos aspectos avanzados de este paradigma y del uso de clases. Vamos a entender el concepto de herencia y sus implicaciones. Gracias a estos conocimientos entenderemos nuevas características del lenguaje Java.

El concepto **herencia** en programación es muy similar al que podemos entender desde un punto de vista de la biología. En el paradigma de programación orientado a objetos, la herencia es un mecanismo que permite la definición de una clase a partir de otra ya existente, lo que permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

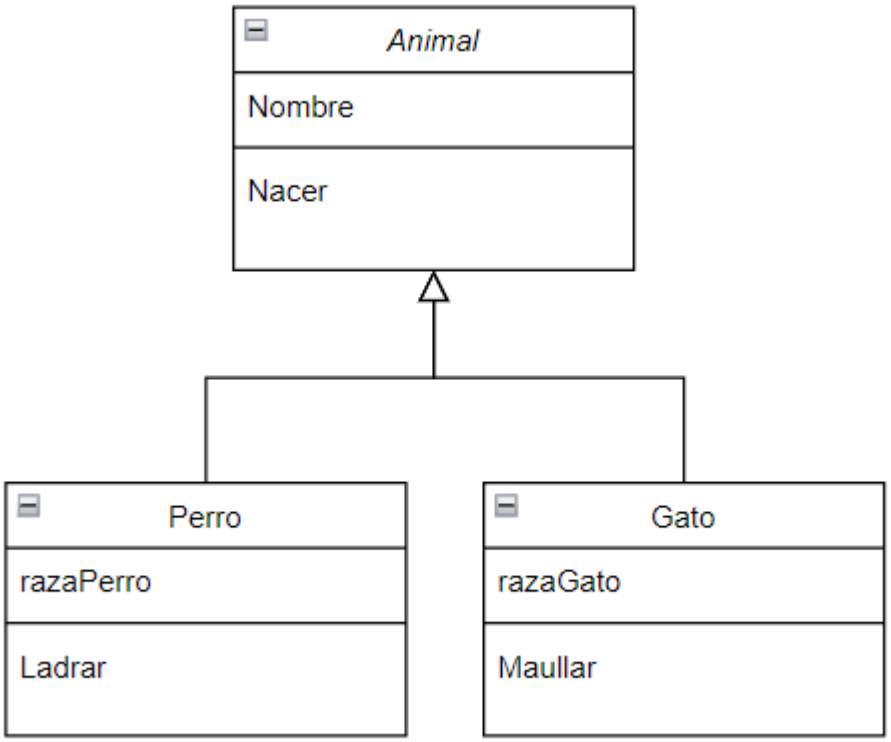
Cuando dos clases tienen una relación de herencia, tendremos la **clase padre o superclase y la clase hija o subclase**. La clase hija podrá heredar elementos de la clase padre.



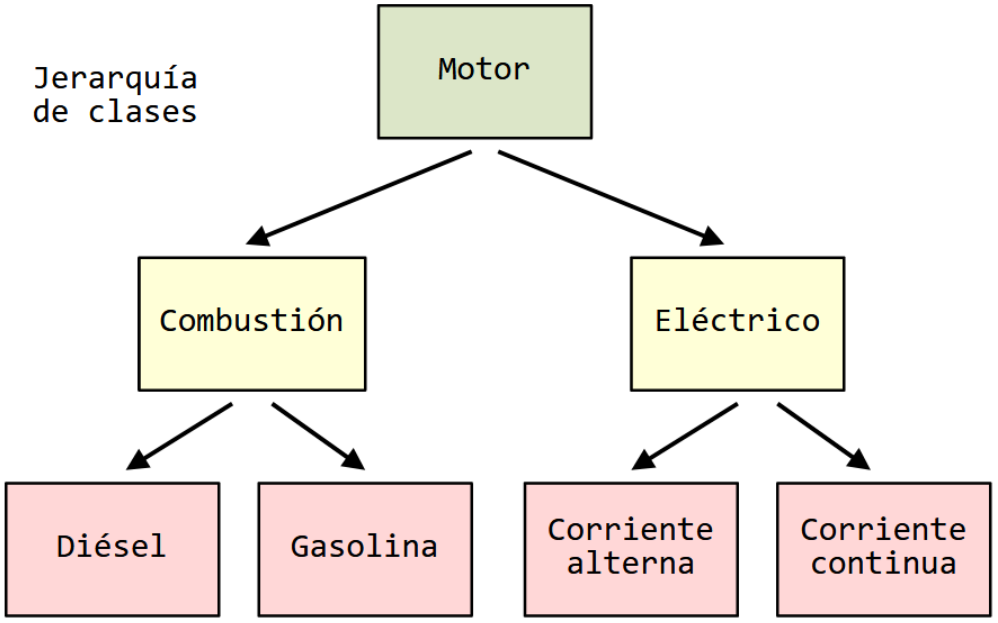
¿Qué ventaja nos aporta tener una clase hija? Esta clase pretende ser una versión especializada de la clase padre que expanda su funcionalidad, ya que la clase hija puede heredar variables y métodos definidos por la clase padre, y además, tener sus propias variables y métodos. Ofrece una gran ventaja a la hora de reutilizar código.



Ten en cuenta que la notación usada para ilustrar estos ejemplos pretende ser clara y sencilla, pero no es una forma habitual de especificar y documentar software. Para ello se usan otros diagramas estandarizados, como UML. Estos diagramas quedan fuera del alcance del módulo de Programación, pero los aprenderás en otros módulos. El siguiente ejemplo representa un diagrama de clases estandarizado UML (presta atención al sentido de las flechas).



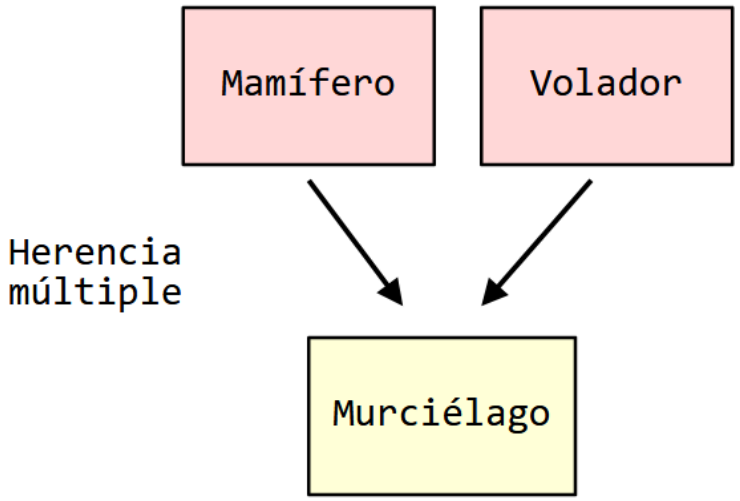
Esta relación superclase-subclase puede extenderse y formar una jerarquía de clases. El diseño de programas o soluciones con jerarquías de clases puede ser complejo, pero aportará a la solución final robustez, calidad, mantenibilidad y eficiencia en el código, que son características muy deseadas en las soluciones de software profesional.



Desde el punto de vista de las **ciencias de la computación** existen dos tipos de herencia:

- **Herencia simple**. Una clase solo puede tener un padre, en consecuencia la estructura de una jerarquía de clases con herencia simple será arborescente.

- **Herencia múltiple**. Cada clase puede tener uno o varios padres, en consecuencia la jerarquía de clases con herencia múltiple será un grafo.



Ten en cuenta que el lenguaje **Java no soporta herencia múltiple**, por ello este lenguaje no puede explotar las ventajas de la herencia múltiple de forma directa, aunque, como veremos en las próximas sesiones de trabajo, Java dispone de otras herramientas para suplir su carencia de herencia múltiple.

¿Cómo usar la herencia en Java?
Para usar la herencia en Java necesitamos conocer nuevas palabras reservadas del lenguaje que nos van a permitir crear clases con relación de herencia.

La primera palabra reservada que debemos conocer es **extends**, esta palabra se emplea en la definición de una clase y nos permite indicar la clase padre de nuestra clase. Su sintaxis es la siguiente:

```
public class Hija extends Padre{
    // contenido de la clase Hija
}
```

¿Y qué hereda? Hereda todo aquello que sus modificadores de acceso nos permitan, es decir, los métodos y atributos que sean public o **protected**.

Código de ejemplo

Ejercicios

Ejercicio 1.
Busca información de otros lenguajes de programación. Encuentra alguno que permita herencia múltiple. ¿Cómo se indica la herencia en otros lenguajes de programación (el equivalente a extends en Java)?

Ejercicio 2.
Prueba a cambiar los modificadores de las clases A, B y C y familiarízate con los escenarios en los que puedes acceder y en los que no.

Ejercicio 3.
Recuerdas el modificador final. Si no lo recuerdas revisa la sección “Otros modificadores” de la unidad de trabajo anterior. Prueba a usar el modificador con la jerarquía de clases A, B y C, emplealo para las clases, los atributos y los métodos.

Ejercicio 4.
Hemos entendido la herencia, pero los ejemplos y ejercicios hasta ahora son muy teóricos y poco interesantes. Así que… intenta buscar un ejemplo de herencia más útil, piensa en ejercicios de secciones anteriores o cualquier otro ejemplo, intenta usar herencia en un programa en el que la herencia aporte valor.

Referencias y ampliación

Clase Object

Contenido

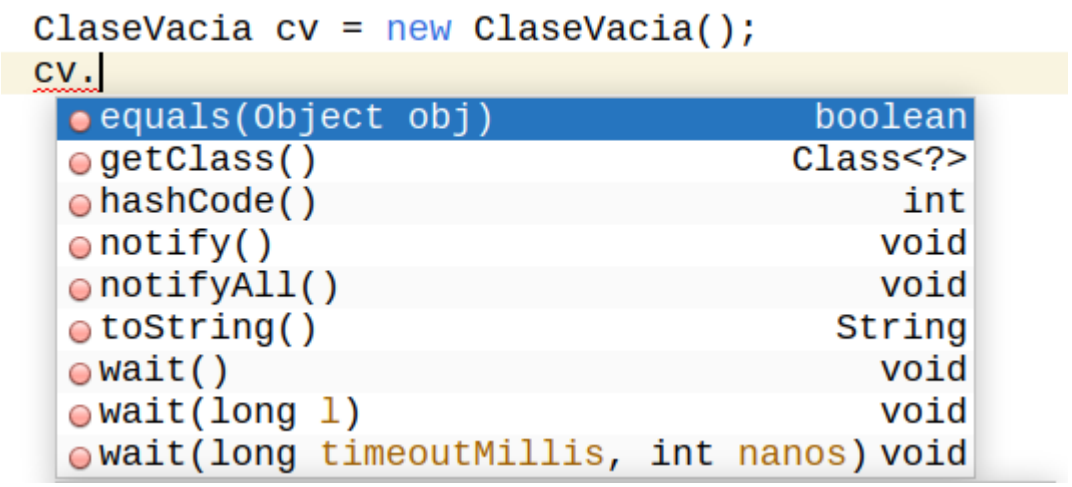
Veamos un ejemplo sencillo. Tenemos la siguiente clase. Está vacía. No tiene parámetros, ni métodos, ni constructor. Además es una clase que no usa la palabra extends para heredar de otra clase.

```java
package ut07e02claseobject;

public class ClaseVacia {

}
```

Pero si vemos qué ocurre en nuestro IDE cuando creamos una instancia de ella, vemos que tiene algunos métodos.



¿Es un error del IDE? No es un error.
¿De dónde vienen estos métodos? Estos métodos pertenecen a la clase Object. Object es una clase de la librería estándar de Java, y además es una clase muy especial, ya que es la primera clase de la jerarquía de clases, el padre de todas las clases.

No hemos heredado de la clase Object, pero Java lo hace por nosotros. ¿Recuerdas lo que ocurre cuando no creas un constructor para una clase? Que la máquina virtual crea para nosotros un un constructor vacío por defecto que no hace nada. Algo similar es lo que ocurre con la herencia. Nosotros podemos crear una relación de herencia mediante la palabra reservada extends, pero si no la usamos, Java lo hará por nosotros, heredando por defecto de la clase Object. Por esto todas las clases heredan características de la clase Object, ya sea porque al no heredar de ninguna otra heredan de Object, o porque alguna de las clases padres de nuestra clase padre en algún momento hereda de Object.

Bien, entonces todas las clases en Java han heredado algo en algún momento de la clase Object. ¿Qué nos aporta Object en nuestras clases? De la clase Object heredaremos algunos métodos interesantes. En próximas sesiones entenderemos más características y propiedades

de estos métodos, pero ahora vamos a ver qué nos aportan algunos de ellos.

Método toString
El método toString pretende transformar en una cadena de texto un objeto. El código del método es algo similar a lo siguiente:

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

Y genera un resultado como este:
ut07e02claseobject.ClaseVacia@6ce253f1

Método hashCode
Cualquier objeto que esté en memoria en algún momento de la ejecución de nuestros programas tiene asociado un código hash único. Este método nos permitirá identificar de forma unívoca un objeto en la memoria, ya que nunca habrá dos has iguales para diferentes objetos.

Algunas estructuras de datos que estudiaremos más adelante en el curso hacen uso de los código hash.

Método equals
Es un método que permite comparar objetos. De forma similar al uso que le damos al operador == para comparar enteros, el método equals nos permite comparar objetos y obtener un resultado booleano (true/false).

Método getClass
Permite obtener la clase de un objeto.

Método finalize
Este método no puede ser llamado desde cualquier parte del código. El método finalize se ejecuta justo antes de que el objeto sea eliminado por el Garbage Collector (el mecanismo de Java para eliminar de la memoria objetos que no van a ser usados o que no son necesarios).

De momento no entendemos las utilidades que puede tener este método, pero se entenderán más adelante en el curso.

Método clone
Este método devuelve un objeto exactamente igual que el que estamos clonando. Este método heredado de la clase Object no es usable directamente, en la siguiente sección se explica cómo puede ser usado.

El resto de métodos de Object (wait, notify y notifyAll) están relacionados con sincronización y concurrencia, que quedan fuera del alcance de este módulo.

Código de ejemplo

Ejercicios

Ejercicio 1.
Usa el enlace a la documentación de Object en la documentación de Java e intenta entender toda la información adicional que seas capaz, presta especial atención a los aspectos que aún no puedes entender en su totalidad.

Ejercicio 2.
Usa el enlace al artículo de la Wikipedia sobre hash e informate en más profundidad sobre este código y cómo se calcula.

Ejercicio 3.
Busca información adicional sobre cómo se pueden usar los métodos clone, equals o toString.

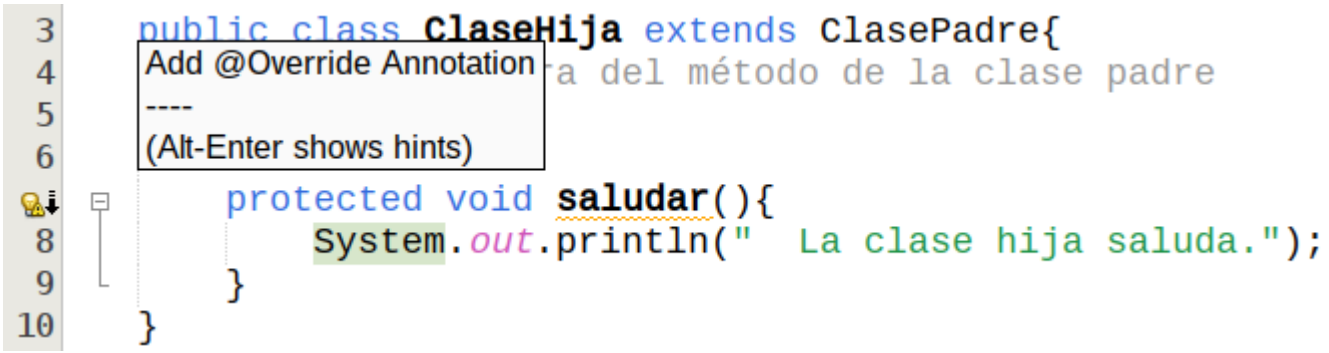
Referencias y ampliación

Sobrescritura

Contenido

La sobrescritura de métodos nos permite cambiar el funcionamiento de los métodos que hemos heredado. Si el funcionamiento de alguno de los métodos heredados no es el deseado lo podemos cambiar para que nuestra clase lo haga según sus necesidades.

Para sobrescribir un método debemos llamarlo en nuestra clase de la misma forma que se llamaba en la clase padre, tendrá los mismos parámetros de entrada y salida, y **nuestro método tendrá que tener el mismo modificador de acceso, o uno más permisivo** que el método original. Si el método que queremos sobreescribir era protected podremos sobreescribirlo como public o protected, pero no private.



Con llamar al método de la misma forma que lo hacemos en la clase padre sería suficiente y funcionaría. Pero esta forma no está recomendada, ya que en caso de que no exista ese método en la clase padre nos dará un error, y así el código es más fácil de mantener y entender. Si queremos sobrescribir un método debemos poner la anotación @Override sobre el método.

```
@Override
protected void saludar(){
    System.out.println("  La clase hija saluda.");
}
```

En ocasiones querremos cambiar el método para nuestra clase, pero manteniendo lo que hacía el método de la clase padre. En este caso, podemos llamarlo mediante super para usar el método original dentro de nuestro método sobrescrito de forma similar a como hacíamos esta operación con los constructores sobrecargados.

```java
@Override
protected void saludar(){
    super.saludar();
    System.out.println("  La clase hija saluda.");
}
```

Código de ejemplo

Ejercicios

Ejercicio 1.
Usa el enlace de anotaciones para entender qué son las anotaciones. Busca más anotaciones diferentes de Java. Busca si otros lenguajes de programación tienen anotaciones.

Ejercicio 2.
Busca más métodos que normalmente se sobrescriban, como el caso de toString.

Ejercicio 3.
Crea una clase Deportista. Y crea tres clases que hereden de ella: tenista, futbolista y ciclista. Tanto la clase deportista, como las tres clases hijas deben sobrescribir el método toString de la forma que consideres. Además, las cuatro clases deben tener un método “hacerDeporte” que indique qué hace cada deportista.

Ejercicio 4.
En el ejercicio anterior ponle el modificador final al método “hacerDeporte” de la clase Deportistas. ¿Qué ocurre? ¿Por qué? ¿Y si en lugar de final pones el modificador static?

Referencias y ampliación

Polimorfismo

Contenido

En esta sección hablaremos del polimorfismo, primero desde el punto de vista de las ciencias de la computación y posteriormente su aplicación a Java.

En las secciones anteriores hemos trabajado con el polimorfismo de forma directa o indirecta, pero es momento de definirlo y valorar sus características.

El polimorfismo es una propiedad que tienen los objetos para comportarse de la misma forma ante diferentes tipos de datos u objetos. Desde el punto de vista de las ciencias de la computación existen diferentes tipos de polimorfismo y diferentes autores tienen diferentes formas de clasificarlos en base a los paradigmas de programación.

En nuestro caso vamos a entender cómo hemos usado el polimorfismo hasta el momento y de qué otras maneras es habitual hacerlo en el lenguaje Java con su paradigma orientado a objetos.

Un tipo de polimorfismo puede ser la sobrecarga que tratamos en la unidad de trabajo anterior, que nos permite obtener un resultado de la misma naturaleza para diferentes condiciones iniciales (diferente número de parámetros.

Otros tipo de polimorfismo lo vemos en la sobreescritura de métodos, la clase padre saluda de una forma y la clase hija de otra, pero ambas pueden responder a la acción de saludar, cada una de ellas de su forma.

En Java, una de las principales implicaciones de esto es que podemos usar un objeto de una subclase en un escenario en el que queremos un objeto de la superclase. Esto es, podremos tratar a todas las clases como cualquiera de las clases que estén por encima en la jerarquía.

```
ClasePadre instanciaPadre = new ClasePadre();
ClaseHija instanciaHija = new ClaseHija();
ClaseHijaSP instanciaHijaSP = new ClaseHijaSP();

// tenemos tres clases diferentes, pero podemos tratarlas como
// iguales desde el punto de vista de su jerarquía

ClasePadre[] array = {instanciaPadre, instanciaHija, instanciaHijaSP};

// hacemos un bucle para que todos los elementos del array saluden
for(ClasePadre p : array){
    p.saludar();
}
```

En este ejemplo las clases hijas han sido capaces de comportarse como la clase padre, tanto almacenadas en una array de padre, como al llamar al método saludar de cada una de ellas.

Entonces, ¿Puedo guardar todo en una clase superior en la jerarquía?

```
Object[] trastero = new Object[5];

trastero[0] = instanciaPadre;
trastero[1] = instanciaHija;
trastero[2] = "Hola";
trastero[3] = 3;
trastero[4] = new Scanner(System.in);
```

Pero esto solo me permitirá usar los métodos de la clase Object. Su funcionalidad dependerá de la situación, pero no es una práctica habitual.

Cuando estamos recorriendo elementos de una jerarquía, como en los ejemplos anteriores, en ocasiones tendremos la necesidad de conocer si un objeto es una instancia de una clase concreta. Java nos permite conocer esto gracias a la palabra reservada **instanceof**. Se puede usar de la siguiente forma:

```
miInstancia instanceof Object;
```

Esta expresión devolverá true/false, en función de si es o no es una instancia de la clase que estamos referenciando.

Código de ejemplo

Ejercicios

Ejercicio 1.
Vamos a crear un programa sencillo que calcule las puntuaciones de dos equipos de baloncesto. Para ellos debes crear los siguientes elementos:

- Un enumerado que indique el tipo de anotación: tiro libre (1 punto), tiro dentro del arco (2 puntos) y triple (3 puntos).
- La clase lanzamiento, que representa que un jugador ha tirado a canasta.
- Las clases lanzamiento de tiro libre, lanzamiento de tiros dentro del arco y la clase lanzamientos de triples.

El programa debe generar dos números aleatorios de lanzamientos para cada equipo (entre 50 y 70 lanzamientos por equipo).
Aleatoriamente, con la misma probabilidad el lanzamiento debe tratarse de cualquiera de los tres tipos.

- Los lanzamientos de tiros libres tienen un 60% de convertirse en puntos anotados.
- Los lanzamientos de tiro dentro del arco tienen un 50% de convertirse en puntos anotados.
- Los lanzamientos de triple tienen un 40% de convertirse en puntos anotados.

El programa debe calcular la puntuación de cada equipo y decidir el ganador. El ejercicio se puede resolver de diferentes formas, pero intenta usar los principios del polimorfismo y los elementos indicados anteriormente.

Ejercicio 2.
Investiga cómo se puede hacer un casting entre clases. Piensa algún ejemplo de uso de casting entre clases en escenarios en los que usemos el polimorfismo.

Referencias y ampliación

Clases abstractas

Contenido

Las clases abstractas son un tipo especial de clase de la que no se pueden crear instancias. Podríamos pensar que son clases que están pensadas para usar sus métodos estáticos, pero no es así. Las clases abstractas están pensadas para ser usadas en la herencia.

Pongamos un ejemplo. Necesitamos poder gestionar todas las personas de un instituto, tendremos alumnos, profesores, invitados y personal auxiliar. Todos son personas y posiblemente tengan algunos métodos comunes, podríamos crear la clase Persona y que de ella hereden todas las demás, pero en este escenario la clase persona podría no tener ninguna utilidad, pero puede ofrecer funcionalidad común que pesea no tener sentido de forma aislada, tiene sentido para todas sus clases hijas.

Para crear una clase abstracta debemos usar el modificador abstract delante de la clase, y además lo podremos usar en los métodos abstractos, como en el siguiente ejemplo:

```
abstract class ClaseAbstracta {
    int variable = 3;

    void informarVariable(){
        System.out.println("El valor de la variable es " + this.variable);
    }

    abstract void metodo1();
    abstract void metodo2(int a);


}
```

Es importante entender que no es necesario que tenga métodos abstractos, puede tenerlos o puede no tenerlos, dependiendo del uso que queramos hacer de la clase. En caso de tener métodos abstractos se crea solamente la definición del método, sin el cuerpo, con el valor retornado, el nombre y los parámetros que recibe.

Las clases que hereden de esta clase abstracta heredarán su funcionalidad y se verán obligadas a sobrescribir los métodos abstractos, en caso de no hacerlo no se podrá compilar el programa y dará un error.

Código de ejemplo

Ejercicios

Ejercicio 1.
¿Puede una clase abstracta tener constructor? ¿Puede heredar de otra clase? ¡Pruébalo y sal de dudas!

Ejercicio 2.
Si tengo una clase abstracta y creo una nueva clase abstracta heredando la anterior, la nueva está obligada a implementar los métodos abstractos? ¿O puede dejarlos como abstractos? ¡A probar!

Ejercicio 3.
¿Puede un método abstracto ser static? Ya sabes lo que toca…

¡Termina los ejercicios por tu cuenta!

¡Aquí no se te ha perdido nada, vuelve arriba!

Venga… te lo digo yo:

- Una clase abstracta puede tener constructor y heredar de otra clase (normal o abstracta).
- Una clase abstracta que hereda de otra clase abstracta puede implementar los métodos heredados o los puede seguir dejando como abstractos.
- Un método abstracto no puede ser estático.

Referencias y ampliación

Interfaces

Contenido

¿Recuerdas la herencia múltiple? En la primera sección de esta unidad de trabajo hablábamos de ella y decíamos que en Java no está permitida. Pero para suplir esta carencia Java tiene una herramienta que permite hacer algo similar a lo que permite la herencia múltiple en otros lenguajes: las interfaces.

Una interfaz es una colección de métodos abstractos y atributos constantes en los que se especifica qué se debe hacer, pero serán las clases hijas las que decidan su comportamiento.

En este punto parece que una interfaz y una clase abstracta son muy similares, la única diferencia es que una clase abstracta puede implementar parte de la funcionalidad y la interfaz no puede implementar nada de la funcionalidad.

Pero la mayor diferencia es que la herencia y el uso de interfaces están separados en Java, puedes heredar de una clase abstracta (o normal) y además implementar la funcionalidad de una interfaz, y no solo eso, puedes forzar a que una clase implemente la funcionalidad de tantas interfaces como quieras. Las interfaces son contratos que vamos a obligar a cumplir a todas las clases que implementen dicha interfaz.

Vamos a desglosar esto con algunos ejemplos. Aquí podemos ver una interfaz, se usa la palabra reservada interface en lugar de clase, y esta no puede tener constructor.

```
public interface Interface1 {
    public void metodo1();
}
```

Y así la usamos en una clase, mediante la palabra reservada implements.

```java
public class ClaseQueUsaIntefaz implements Interface1 {
    int i;

    public ClaseQueUsaIntefaz(int i){
        this.i = i;
    }

    @Override
    public void metodo1(){
        System.out.println("Hola soy el método1");
    }

}
```

Veamos algunas características importantes de las interfaces:

- No puedes tener constructores.
- Pueden incluir atributos con el modificador final.
- Pueden incluir métodos con cuerpo que implementen una funcionalidad, tanto estáticos como default.
    - Los métodos estáticos son lo que ya conocemos.
    - Los métodos default son métodos que aportan funcionalidad de base, y que podrán ser sobrescritos en caso de necesidad por las clases que implementen las interfaces. (ver nota al final)
- Los atributos y los métodos sin cuerpo deben ser de tipo public, aunque no indiquemos el tipo este será public.
- **Se puede hacer referencias a los objetos como instancias de la interfaz que usan.**
- Una clase puede implementar más de una interfaz.
- Las interfaces pueden heredar de otras interfaces, y en este caso sí que se permite la herencia múltiple.

Tras entender la funcionalidad de las clases abstractas y de las interfaces, es habitual tener dudas sobre cuándo usar cada una. Esto siempre dependerá de la solución que estemos diseñando y de las diferencias que existen entre interfaces y clases abstractas, pero aquí se proponen algunas consideraciones que pueden ayudar a decidir cual usar.

Clases abstractas cuando se cumpla alguna de las condiciones:

- Quieres compartir código entre muchas clases relacionadas.
- Esperas que las clases que extiendan tu clase abstracta tengan en común métodos o campos.

- Quieres disponer de atributos no estáticos o no finales. Esto te habilita para definir métodos que puedan acceder y modificar el estado del objeto al que pertenecen.

Interfaces cuando se cumpla alguna de las condiciones:

- Esperas que clases sin relación entre sí implementen tu interface.
- Quieres un comportamiento específico sin importarte la implementación.
- Quieres disponer de herencia múltiple.

La palabra reservada **instanceof** puede utilizarse con interfaces, al igual que con clases. Así si una clase está implementando una interfaz, obtendremos true tanto si preguntamos por la clase como si preguntamos por la interfaz.

Nota: A partir de la versión 8 de Java se introdujo el método default para las interfaces. Esta funcionalidad no está disponible en versiones anteriores de Java.

Código de ejemplo

Ejercicios

Ejercicio 1.
Entiende y analiza todo el código adjunto de esta lección. Realiza modificaciones para familiarizarte con su uso.

Ejercicio 2.
Crea una interfaz para una figura geométrica, luego crea las clases cuadrado, triángulo y círculo que implementen dicha interfaz. Genera aleatoriamente entre 5 y 10 elementos de cada tipo, almacenarlos todos en un array y luego recorre el array para que todos los elementos, independientemente de su clase, digan cual es su área y su perímetro.

Referencias y ampliación

UT08 - Control y manejo de excepciones

Errores y excepciones. Jerarquía. Captura.

Contenido

Cuando escribimos un programa en Java pueden ocurrir diversos problemas que no permitan ejecutar dicho programa, o que permitan ejecutarlo de forma incorrecta.

¿Qué problemas nos vamos a encontrar?

1. Bugs. Los bugs son un tipo de error que representa un mal funcionamiento del programa desde el punto de vista de los requisitos del mismo. Imagina que necesitas hacer un programa que cuente del 1 al 10, pero tu programa cuenta del 1 al 12. Esto es un bug, ya que supone un comportamiento indeseado, pero desde el punto de vista de la lógica de programación está todo correcto, Java no puede saber si tu querías contar hasta 10 o hasta 12.
2. Errores. Los errores suelen estar vinculados a una mala escritura del lenguaje Java, por ejemplo no poner un ; al final de una instrucción o usar una palabra reservada del lenguaje como nombre de una variable. Estos errores no permiten que el programa comience a ejecutarse. Es lo que

solemos ver subrayado en rojo en los IDE de programación. Si no lo solventamos no podremos compilar el código, ya que el compilador no entenderá qué queremos hacer ya que para él las instrucciones no son entendibles.

3. Excepciones. Las excepciones son errores que se producen en el momento de la ejecución del programa. Nuestro programa no tiene errores, con lo cual se puede ejecutar, pero en mitad de dicha ejecución surge un problema y el programa debe finalizar abruptamente.

En esta unidad de trabajo nos vamos a centrar en las excepciones.

En la unidad de trabajo 3 usamos las excepciones sin tener un conocimiento completo de ellas. Si las escribíamos tal y como nos lo indicaba el profesor podíamos evitar problemas en el uso de la clase Scanner. En este momento del curso, gracias a los conocimientos adquiridos de POO estás en posición de entender plenamente su funcionamiento.

Y es que, como todo en Java, las excepciones son objetos, y usan algunas propiedades de estos que son relevantes para entender su funcionamiento.

¿Las excepciones no se pueden prever cuando estoy programando?
Algunas sí y otras no.
Algunas excepciones ocurren por problemas en la programación. Por ejemplo, si he creado un array de 10 números enteros, y posteriormente intento acceder a la posición 15 de dicho array, se generará una excepción, ya que esta posición no existe, pero el programa compila y comienza a ejecutarse.

Otras excepciones ocurren por problemas externos al programa. Por ejemplo, una parte de mi programa debe conectarse con un servidor para preguntar por un dato, pero el cable de red del ordenador en el que se está ejecutando el programa está desconectado. Es un problema para el funcionamiento del programa, pero no está en la mano del programador que dicho cable esté conectado correctamente.

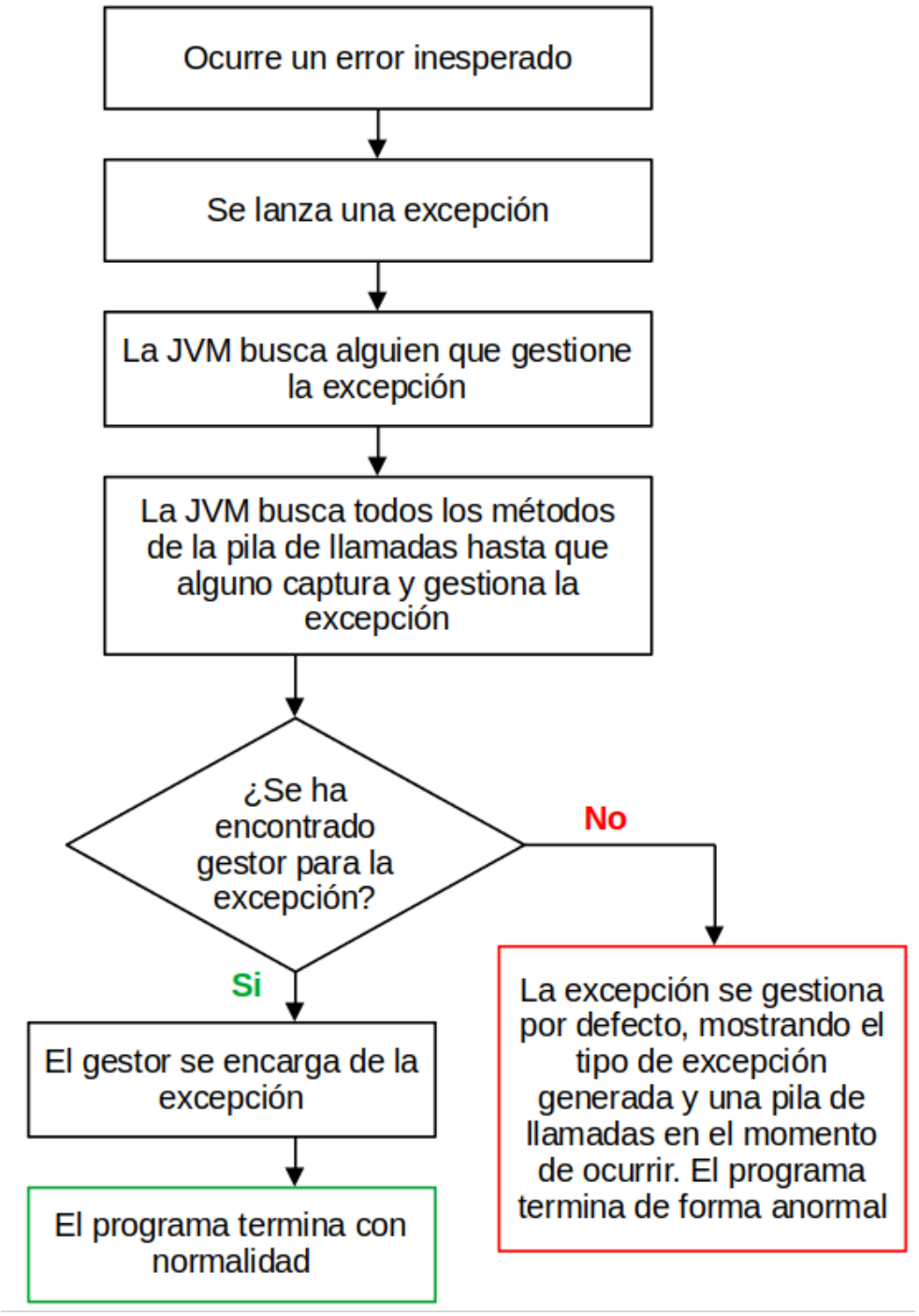
Lo que siempre está en la mano del programador es prever cuándo puede ocurrir una excepción y adelantarse a ella gestionando el problema en caso de ocurrir.

Algunos ejemplos de excepciones comunes pueden ser las siguientes:

- Intentar abrir un archivo que no existe.

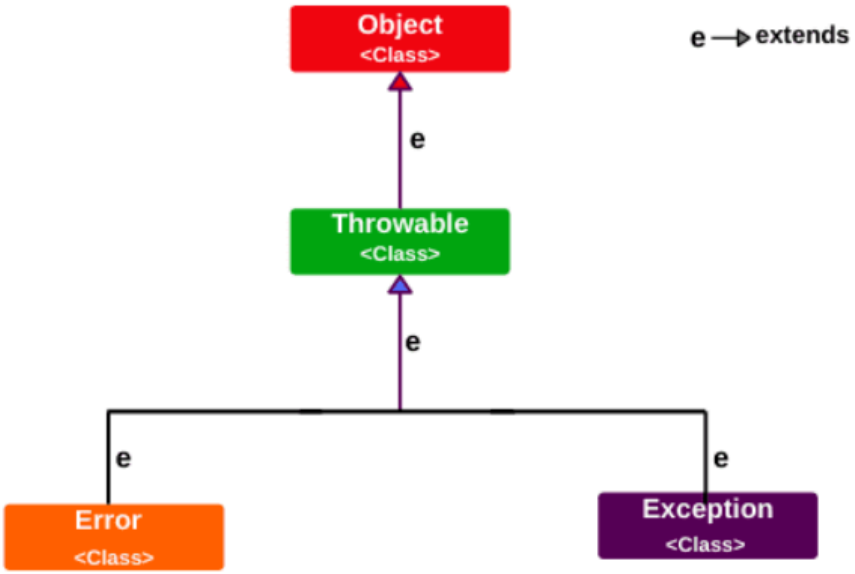
- Hacer una división por 0.
- Preguntar al usuario un número pero que responda con otro tipo de dato.
- Intentar acceder a un índice de un array que está fuera de su tamaño.

¿Qué ocurre cuando se produce una excepción en un programa Java?



Jerarquía de errores
Los errores y excepciones, como todo en Java, son una clase, y en consecuencia forman parte de una jerarquía de clases.

En la siguiente imagen vemos la jerarquía de la que proviene la clase exception.



Tanto Exception como Error son clases que extienden de una clase llamada Throwable y emplearán algunas de sus propiedades heredadas.

No estamos muy acostumbrados a encontrarnos con errores de tipo Error, puesto que los IDE nos ayudan a predecirlos gracias al análisis de nuestro código y predicen los errores de forma similar a cómo funcionan las correcciones ortográficas en un procesador de textos como Microsoft Word o LibreOffice Writer.

Existen otros tipos de errores de los cuales no se puede recuperar un programa, como los causados por desbordamiento, errores de enlazado, accesos no permitidos a memoria, etc.

En la siguiente imagen podemos ver algunas excepciones que dependen jerárquicamente de la clase Exception.

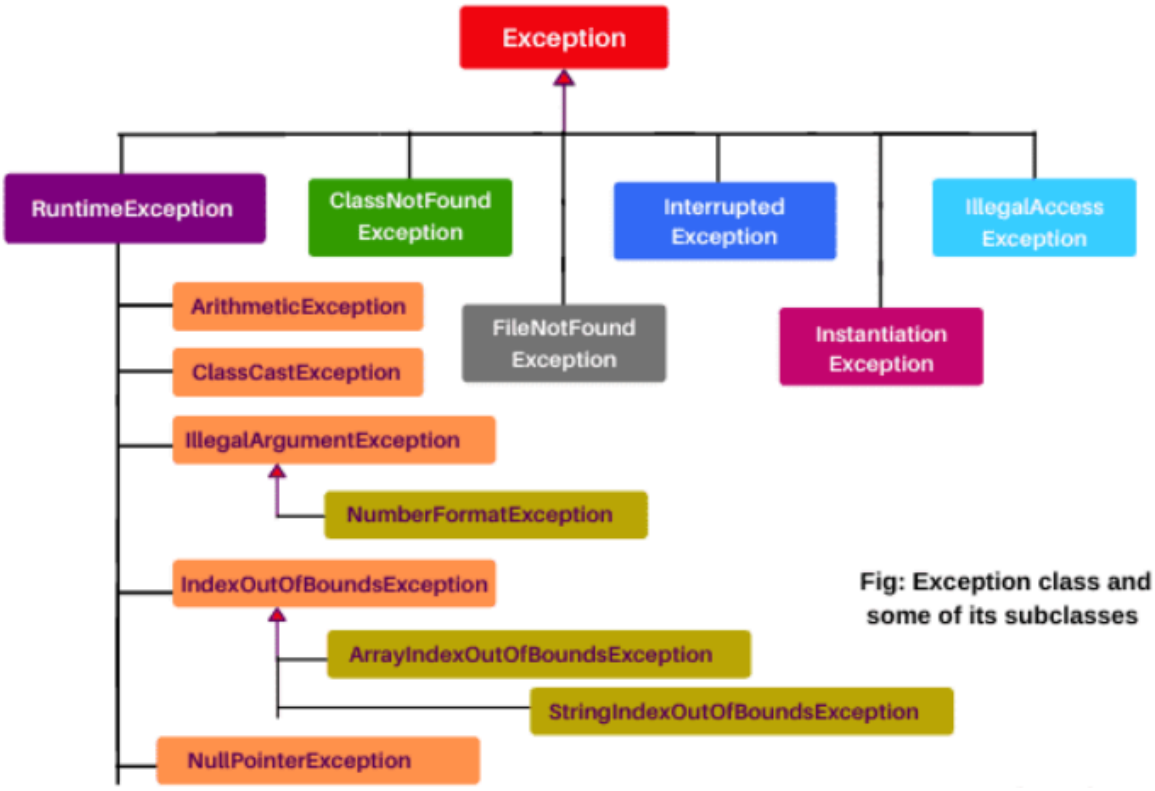


Fig: Exception class and some of its subclasses

Excepciones comprobadas y no comprobadas
En Java existen excepciones comprobadas (checked) y no comprobadas (unchecked).

Las excepciones comprobadas son aquellas que el compilador de Java va a comprobar si están siendo gestionadas en el código, proponiendonos que las gestionemos (el IDE nos indicará que debemos gestionar el código que pueda generar excepciones comprobadas).

Las excepciones comprobadas son aquellas que pueden ocurrir, pero no serán analizadas por el compilador, si no por la propia JVM.

Las excepciones no comprobadas son las de tipo RuntimeException o subclases de esta. Las demás excepciones son todas de tipo comprobadas.

Código de ejemplo

Ejercicios

Ejercicio 1.
¿No estás del todo familiarizado con qué es una pila de llamadas? Explora este artículo para entender su funcionamiento en Java.

Ejercicio 2.
Crea programas con errores (subrayados en rojo en el navegador), intenta ejecutarlos y ver el mensaje de error.

Ejercicio 3.
Explora el enlace a la documentación del API de Java que muestra la clase Exception, identifica su jerarquía y sus métodos.

Referencias y ampliación

Captura de excepciones

Contenido

Tal y como vimos en la unidad de trabajo 3, para capturar excepciones debemos emplear un bloque de código try-catch-finally.

¿Cómo usamos este bloque de código?

```
try {
    // aquí ponemos el código que puede causar la excepción
} catch (Exception e) {
    // aquí ponemos el código que se ejecutara si se produce
    // la excepción, si no se produce la excepción este código
    // no se ejecuta
} finally {
    // aquí ponemos el código que se ejecutará tanto si se
    // produce la excepción como si no se produce
}
```

En el bloque catch ponemos el código que se ejecuta cuando se produce la excepción que ponemos entre paréntesis tras el catch. A esa instancia de la excepción le damos un nombre (habitualmente e) y

sobre ella podremos usar los métodos que heredamos de esta jerarquía de clases.

El bloque finally es opcional, podemos usarlo o no usarlo. Un ejemplo típico de uso es el de optimizar recursos. Por ejemplo, con una instancia de la clase Scanner. Cuando nosotros creamos un Scanner este consume recursos del sistema hasta que lo cerramos.

Puede parecer que usar el bloque finally y no usarlo es igual, ya que podemos hacer estas operaciones de limpieza fuera del bloque try-catch sin un finally, y en la gran mayoría de los casos eso es cierto. Si dentro de try o catch se produce un final abrupto, por el uso de continue, break o return, el bloque finally se ejecutará de todos modos.

Concretando las excepciones

En la lección anterior aprendimos que las excepciones se organizan en una jerarquía de clases. Teniendo en cuenta esto, si la excepción que capturamos siempre es “Exception”, estamos haciendo el equivalente a guardar todo en un array de Objects, y esto es una mala práctica, porque no concreta y especializa el código.

Debemos intentar capturar la excepción que prevemos, ya que esto puede ayudarnos a controlar mejor nuestro código y a ofrecer a los desarrolladores más información sobre los problemas ocurridos.

Así, en el caso de Scanner, será más conveniente usar este catch:
catch(InputMismatchException e)

Capturando múltiples excepciones

Pero, qué ocurre si pretendo detectar más de una excepción. En este caso, poner dentro del catch una excepción de tipo “Exception” es una buena solución ya que independientemente del tipo de excepción producida, esta será capturada.

Pero existe una solución mejor. La captura de múltiples excepciones en un solo bloque, con diferentes catch. Es una estructura similar a la que usamos con la estructura if else, que podemos poner diferentes niveles de if.

```java
try {

    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
} catch (NullPointerException e) {
    System.out.println(e.getMessage());
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Código de ejemplo

Ejercicios

Ejercicio 1.
Investiga qué excepciones pueden ocurrir al usar la clase Scanner y realiza un programa en el que se gestionan todas las excepciones de la clase Scanner en una función.

Ejercicio 2.
Busca ejemplos en internet (StackOverflow y sitios similares) de uso de finally. ¿Qué tecnologías o librerías suelen usar las excepciones?

Ejercicio 3.
Modifica el ejercicio 1 para que todas las excepciones que pueden ocurrir con la clase Scanner se mitiguen mediante código sin llegar a generar la excepción.

Referencias y ampliación

Lanzar y propagar excepciones

Contenido

En la primera lección de esta unidad de trabajo decíamos que la JVM buscará algún método que gestione la excepción, pero hasta ahora siempre hemos visto que la excepción se gestiona en el mismo punto del programa en el que se produce, mediante el uso de la estructura try-catch-finally.

Además, las excepciones que se han producido en nuestros programas hasta ahora son las que produce el sistema. En algunas ocasiones queremos lanzar excepciones que no sean generadas por el sistema.

En esta lección vamos a entender el uso de dos nuevas palabras reservadas y cómo estas gestionan el manejo de las excepciones.

Throw (generar una excepción desde el código)
Hasta ahora, si queríamos generar una ArithmetiocException, teníamos que hacer alguna operación que dividiese por 0, por ejemplo:

```
int i = 1/0;
```

Esta instrucción genera una excepción por sí sola, ya que el resultado de dividir por cero es una indeterminación que no puede resolver el sistema.

Pero, ¿y si en nuestro programa que una variable tenga el valor 20 puede causar un problema aritmético?

En este caso podemos lanzar nosotros las excepción cuando queramos mediante la palabra reservada throw, como en los siguientes ejemplos:

```
// creando un objeto del tipo de la excepción y luego lanzándolo
ArithmeticException miExcepcion = new ArithmeticException();
throw miExcepcion;

// lanzándolo directamente
throw new ArithmeticException();
```

Este tipo de objetos pueden ser lanzados ya que heredan de la clase Throwable.

Throws (transferir una excepción en lugar de tratarla)
La palabra reservada throws se emplea en la cabecera de un método, como en este ejemplo:

```
public static void lanzoExcepcion() throws ArithmeticException{
      int i = 1/0;
}
```

Si en la cabecera del método hacemos esta declaración lo que estamos haciendo es transferir la excepción generada al punto del código desde el que se ha llamado a esta función, y allí se encargarán de tratarla.

Siempre que se genera una excepción debemos tratarla, pero podemos hacerlo de dos formas: mediante un bloque try-catch o mediante throws.

Una característica importante de esta palabra reservada es que se pueden lanzar múltiples excepciones, como en este ejemplo:

```
public static void lanzoExcepciones() throws ArithmeticException,
NullPointerException, IOException{
      // código del método
}
```

Código de ejemplo

Ejercicios

Ejercicio 1.
Modifica el código del ejercicio 1 de la lección anterior para que en lugar de gestionar la excepción en el método en el que se causa, sea gestionada en el método que lo llama.

Referencias y ampliación

Excepciones personalizadas

Contenido

¿Qué ocurre cuando la API de Java no tiene una excepción que se ajuste a las necesidades de lo que nosotros consideramos un error para nuestro programa?

En este caso podremos crear nuestras propias excepciones mediante herencia. Como en el siguiente ejemplo:

```
public class MiExcepcion2 extends Exception{

	public MiExcepcion2() {
	}

	public MiExcepcion2(String info) {
	super(info); // pasamos info al constructor de Exception
	}

}
```

Lo más habitual es tener al menos dos constructores, el constructor vacío y un constructor con un mensaje que se lo podemos pasar al constructor de la clase padre para aprovechar los métodos que esta nos presta.

Se puede heredar de Exception, o de cualquier otra excepción existente en el API de java como IOException o RuntimeException. También puedes heredar de tus propias excepciones.

Código de ejemplo

Ejercicios

Ejercicio 1.

Crea una clase Persona con atributos nombre y edad. Crea un nuevo tipo de excepción que se genera cuando una persona tiene una edad imposible (supongamos que la edad posible es entre 0 y 150 años).

Ejercicio 2.
Crea un método para votar en las elecciones. Pero si la persona no es mayor de 18 años debe generar una excepción personalizada.

Referencias y ampliación

UT09 - Colecciones de datos

Colecciones en Java

Contenido

¿Qué son las colecciones?
Las colecciones son unas estructuras de datos de Java que permiten almacenar y manipular objetos. Hasta ahora en el curso hemos usado una colección de Java llamada ArrayList, esta colección nos ayudó a trabajar con objetos mientras aprendíamos los conceptos esenciales de la programación orientada a objetos y la herencia.

En esta unidad de trabajo vamos a profundizar sobre las colecciones, veremos qué problemas resuelven, qué características tienen y cómo se relacionan con otras entidades del lenguaje.

Si pensamos en ArrayList, hasta el momento es una estructura que hemos usado de forma similar a un Array, además sus nombres son muy similares. La parte más interesante de ArrayList frente a Array es lo práctico que resulta emplearla sin necesidad de establecer su tamaño, ya que en muchas situaciones del desarrollo de software no conoceremos el tamaño de los datos que debemos guardar.

Aquí puedes ver la información de ArrayList en la API de Java: ArrayList (vamos a estrenar nuestro nuevo conocimiento de

programación orientada a objetos para leer con detenimiento la información que proporciona la API)

Existen muchas otras colecciones en Java, estas se encuentran dentro de java.util, todas tienen sus ventajas y desventajas, y conocerlas es importante para usar la mejor herramienta en cada escenario.

Es importante recordar que las colecciones sólo pueden almacenar objetos, y no tipos nativos del lenguaje como (int, boolean o double), si necesitamos usar estos datos usaremos sus clases envoltorio (Integer, Boolean, Double).

¿En qué ámbitos se usan?
Las colecciones se emplean en casi todos los ámbitos de la programación. En Java es muy habitual que cualquier programa haga uso de estas. Todos los tipos de aplicaciones se benefician de esta funcionalidad: aplicaciones web, aplicaciones de análisis, de gestión, de dispositivos móviles, etc.

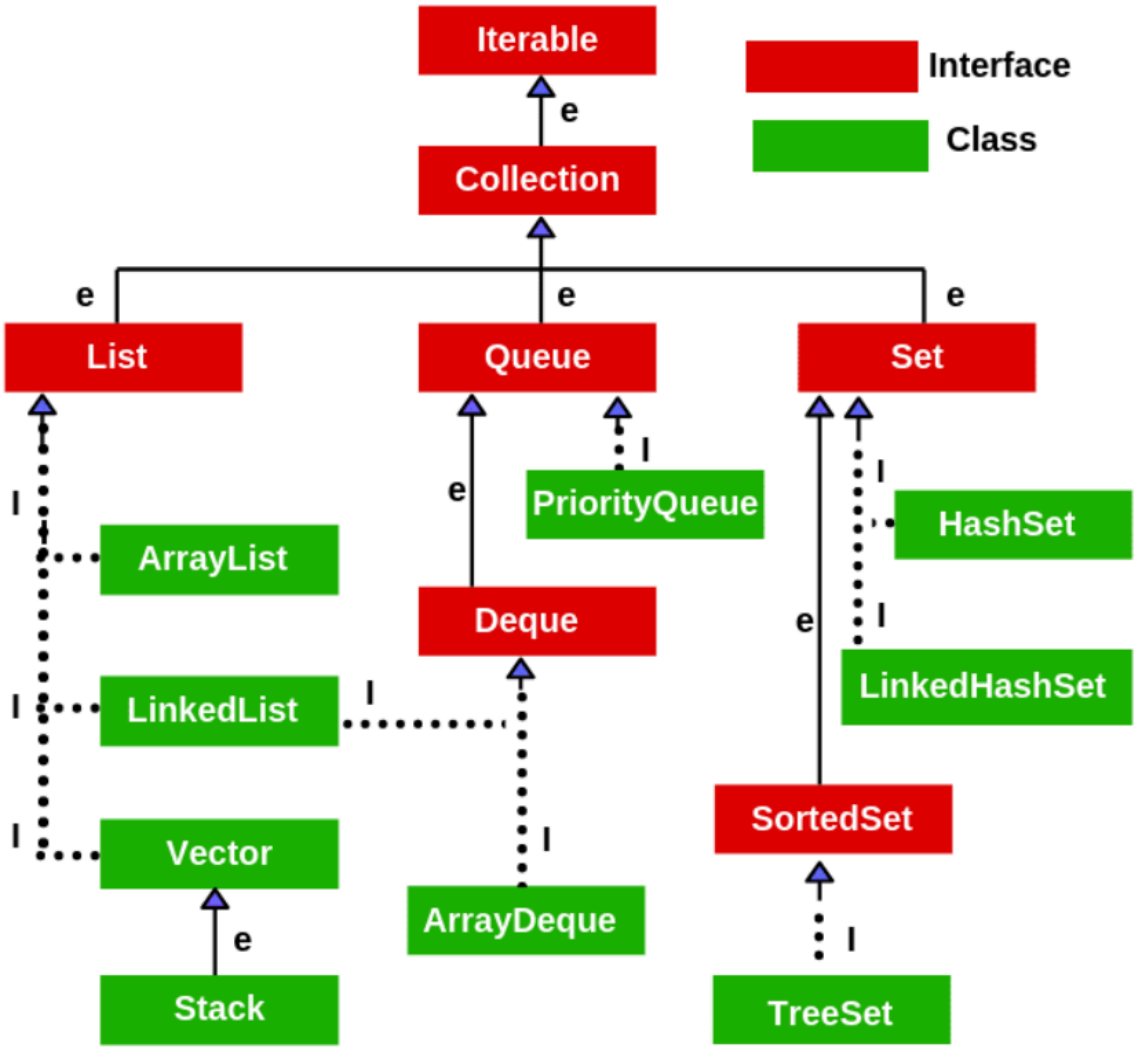
Las colecciones son muy útiles porque resuelven problemas muy comunes en el mundo del desarrollo, su uso no es obligatorio en casi ningún caso, pero muy conveniente en la mayoría. Veamos un ejemplo, imagina que en el lenguaje Java no existiese la clase String, todo lo tendríamos que gestionar con el tipo de datos char. Podríamos hacerlo, imprimir frases no es más que concatenar una secuencia de chars. El problema que se plantea aquí es que el uso de cadenas de caracteres es muy habitual en el mundo de la programación, por ello es muy conveniente tener una clase que nos resuelva la mayoría de problemas de desarrollo que implican su uso. Con las colecciones ocurre algo muy similar.

En próximas lecciones profundizaremos más en las ventajas y formas de usar colecciones, pero de momento podemos saber que nos ayudan a tareas habituales como:

- Almacenar elementos comunes en los que importa el orden en los que han sido almacenados.
- No me aportan utilidad los elementos repetidos.
- Poder saber quien es el siguiente y el anterior a partir de una posición.
- Necesidades de velocidad o tipo de acceso a los datos.
- Necesidades de sincronismo. ¿Qué pasa si dos programas deben leer los mismos datos?
- etc.

¿Cuál es su jerarquía?

Las colecciones siguen una jerarquía de clases como pasa con el resto de clases de Java. Lo que se conoce como el framework de colecciones está compuesto por una serie de clases e interfaces con una relación jerárquica que permite heredar diferentes propiedades. En la siguiente imagen se muestra una jerarquía (no completa) de las colecciones en Java.



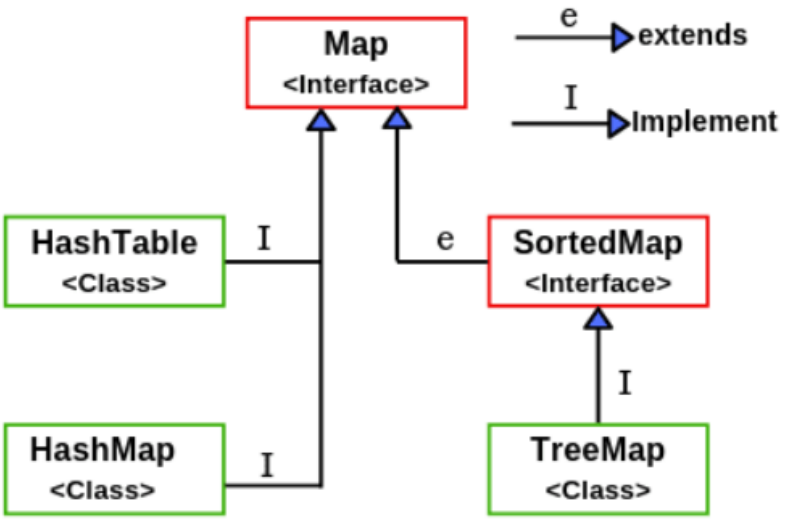
Entendiendo esta jerarquía podremos consultar la API y entender qué características van heredando entre ellas, para conformar diferentes tipos de colecciones.

La interfaz Collection es la raíz principal de la que se heredan las operaciones habituales como añadir o eliminar elementos.

La interfaz Iterable impone la implementación del método iterator(), que entenderemos en próximas lecciones.

List, Queue y Set son las tres interfaces principales que heredan de Collection, y generan las tres grandes familias de colecciones.

Aparte de esta jerarquía, existe otra pequeña jerarquía relevante en las colecciones, que son las que heredan de la interface Map. Estas colecciones usan el concepto de clave-valor, no permitiendo el duplicado de claves. Funcionan como un diccionario, siendo la palabra la clave, y la definición el valor.



Código de ejemplo

Ejercicios

Ejercicio 1.
Empleando la documentación del API de Java. Investiga para qué vale alguna de las colecciones que no se emplean en el código de ejemplo de esta sección. Lo importante de este ejercicio es que seas capaz de entenderlo usando solo la información que te da la API, intente entender la estructura, cómo se muestra el contenido y cómo se vincula con otros conceptos.

Ejercicio 2.

Investiga para qué vale el método peek en una pila.

Ejercicio 3.
¿Investiga qué diferencia hay entre estas dos expresiones?
Queue<String> cola = new LinkedList<>();
Queue<String> cola = new LinkedList<String>();

Ejercicio 4.
Busca información sobre “estructuras de datos” en el ámbito de las ciencias de la computación. Intenta entender qué implica este concepto, qué problemáticas estudia y familiarízate con algunos términos. Para ampliar puedes ver algunos vídeos que expliquen diferentes estructuras de datos, para qué se usan, para qué operaciones son eficientes, etc.

Referencias y ampliación

Genéricos en Java

Contenido

Los tipos genéricos son una solución que ofrece Java para gestionar objetos que no tenemos definidos en el momento de desarrollar, y posteriormente parametrizar dichas clases.

Veamos un ejemplo de instancia de objetos convencional y otra usando genéricos:

```
InfoInteger ii = new InfoInteger(234);
```

En este primer ejemplo tenemos el nombre de la clase (InfoInteger), el nombre que queremos dar a la instancia de la clase (ii) y su invocación al constructor (new …).

```
Info<Integer> ii2 = new Info<>(7);
```

En este segundo ejemplo tenemos el nombre de la clase (Info), la parametrización indicando qué tipo de clase parametrizada nuestra instancia (<Integer>), el nombre que queremos dar a la instancia de la clase (ii2) y su invocación al constructor (new …).

El uso de genéricos tiene algunas ventajas, como la no duplicación de código y la capacidad para establecer tipados en función de la

necesidad, esto hace que nuestros programas sean menos propensos a generar errores de tipo y evita el uso intensivos de casting de objetos.

¿Cómo se traslada esto a la clase Info?
Es muy sencillo, simplemente debemos indicar el parámetro, o los parámetros en la definición, y usarlos dentro del cuerpo de la clase como si fuesen tipos de nuestro programa.

```
public class Info<T> {
    private T dato;

    public Info(T dato){
        this.dato = dato;
    }
```

T es el parámetro, y T dato es una instancia de T. Es importante entender que T no es nada y solo tomará un valor cuando parametricemos la instancia. Así al pasar <Integer> es el equivalente a reemplazar las líneas:
private T dato -> private Integer dato
public Info(T dato){ -> public Info(Integer dato){
En esencia reemplazamos el valor T por la clase que pasemos, esta clase podrá ser propia del lenguaje o una clase que nosotros hemos creado, por ejemplo:
Info<Alumno> ia;

Las clases pueden tener más de un parámetro, por ejemplo:
public class MiClase<T1, T2, T3>

T es un nombre que se suele dar por convenio a estos parámetros genéricos, pero podríamos poner cualquier nombre, al igual que hacemos con las variables.
Para que la legibilidad de los programas sea sencilla por convenio se usan los siguientes valores con genéricos:
E - Element
K - Key
N - Number
T - Type
V - Value

Genéricos de una jerarquía (bounds)
Si queremos que nuestros genéricos no puedan ser cualquier clase, pero si los pertenecientes a una jerarquía podemos hacerlo de la siguiente forma:

```
public class MiClase<T extends Animal>
```

Esto hará que mi clase pueda ser parametrizada con la clase Animal o cualquiera de sus subclases (Perro, Gato, Lagartija, etc.). Además, esto hará que nuestras instancias de T tengan disponibles todos los métodos de la clase Animal, como esperaríamos en una situación de herencia convencional.

También podemos hacer esta herencia con interfaces (se sigue usando la palabra extends, no se cambia por implements):

```
public class MiClase<T extends MiInterfaz>
```

Métodos genéricos

Java nos permite usar los genéricos para que sean usados en un método, en lugar de una clase, por ejemplo, podríamos crear en la clase que contiene el main de nuestro programa un método estático, como en los siguientes ejemplos:

```
public static <T> void miMetodo(T dato){
public static <T1, T2> T1 miMetodo2(T1 dato, T2 dato2){...
```

El primer ejemplo recibe un T y devuelve un void.
El segundo ejemplo recibe un T1 y un T2 y devuelve un T1.

Esto permitirá que llamemos al método con instancias de diferentes clases.

Comodín “?” (Wildcard)

Los genéricos permiten el uso de un operador que se conoce como wildcard, este es el operador “?”. Podemos usar este operador para indicar un parámetro que es una clase que recibe una parametrización.

```
public static void miMetodo(ArrayList<?> lista){...
```

Los comodines pueden usar una jerarquía, al igual que la definición de genéricos en una clase.

```
public static void miMetodo(ArrayList<? extends Animal> lista){...
```

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea un método estático que pueda calcular la nota media a partir de las notas pasadas en un ArrayList. Las clases usadas para inicializar el ArrayList pueden ser Integer, Float o Double.

Ejercicio 2.
Investiga cómo puedes hacer que una clase se parametrice con una clase y dos interfaces.

Referencias y ampliación

¿Qué colección usar para cada situación?

Contenido

El framework de colecciones en Java tiene muchas clases que implementan diferentes interfaces, esto permite tener una gran variedad de herramientas que podemos usar en muchos problemas diferentes de programación.

Pero, ¿cómo sabemos qué tipo de colección usar para cada problema? Para saber qué colección usar debemos conocer qué hacen las colecciones, qué métodos tienen, y sus casos de uso más comunes. El framework de colecciones nos ofrece una gran variedad, y no es objetivo de este curso entender todas ellas en profundidad, pero haremos una aproximación a las características generales de ellas en función de las interfaces que implementan.

La primera separación que podemos hacer de las colecciones es por las interfaces que normalmente usan en sus nombres, por ejemplo ArrayList (de la interfaz List), TreeSet (de la interfaz Set) o HashMap (de la interfaz Map). Conocer la filosofía de funcionamiento de la interfaz List nos va a permitir entender de forma general todas las colecciones de tipo lista, lo mismo es aplicable para las demás.

A continuación explicaremos algunas características principales de las interfaces principales del framework de colecciones de Java.

List
La principal característica de las listas en java es que permiten almacenar elementos en **orden secuencial**. Los elementos se mantienen por índice de posición, este orden se define por su orden de inserción (el primer add, segundo add, tercer add, etc.).

Las listas **permiten tener elementos duplicados**.

Algunas de las **ventajas** del uso de listas son:
- Son de acceso aleatorio (puedo leer cualquier elemento de la lista).
- Son ordenadas.
- Permiten añadir y eliminar elementos.
- Funcionan de forma similar a los arrays, lo que permite usarlas en la mayoría de casos de uso clásicos de arrays.

Algunas de las **desventajas** del uso de listas son:
- Tienen un bajo rendimiento computacional. Si queremos hacer programas óptimos o funciones que se van a ejecutar muchas veces es posible que pueda afectar al rendimiento.

Set
Los set en Java son una colección **desordenada** de elementos. Si añadimos un elemento repetido a nuestra colección este no se guardará por segunda vez.

Algunas de las **ventajas** del uso de los sets son:
- No permite elementos duplicados.

Algunas de las **desventajas** del uso de los sets son:
- No tiene acceso aleatorio.
- Poca eficiencia para ordenar sus elementos.

Map
Los mapas en Java son colecciones que permiten almacenar pares de elementos clave-valor, donde la clave hace la función de índice. Es importante recordar que las colecciones siempre almacenan objetos, no tipos primitivos, en el caso de los mapas, tanto la clave como el valor deben ser instancias de objetos, si necesitamos, por ejemplo, que la clave sea un número lo podremos hacer mediante una clase wrapper, como Integer.

Algunas de las **ventajas** del uso de los mapas son:
- Asociación clave-valor.
- No permiten claves repetidas.

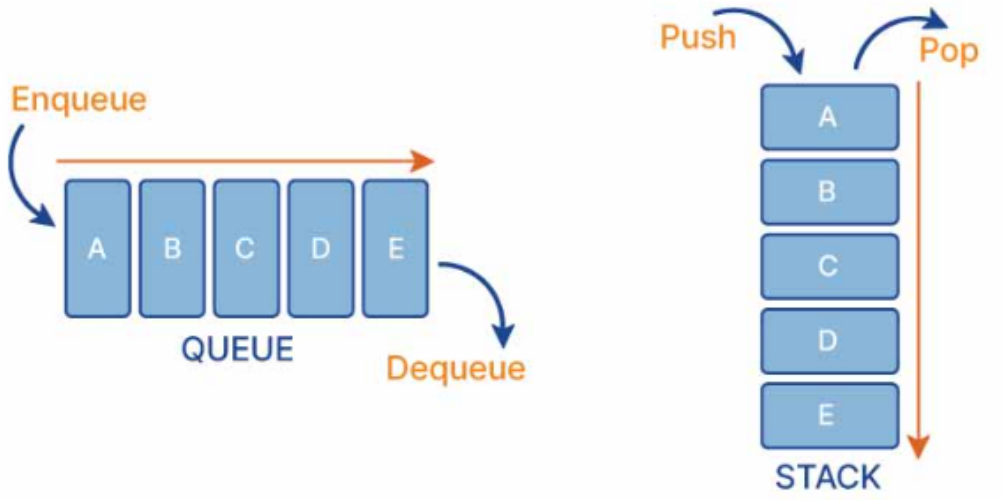
Algunas de las **desventajas** del uso de los mapas son:

- Es una colección muy poco eficiente, incluso menos eficiente que List o Set.

Queue

Las colas en Java representan el funcionamiento clásico de “el primero en entrar es el primero en salir”, este término, en ciencias de la computación se conoce como FIFO del inglés (first in first out). La forma más sencilla de entenderlo es una cola en la caja de un supermercado, la primera persona que se puso en la cola será la primera en ser atendida.

Además de la interfaz Queue, tenemos la interfaz **Deque**, o también conocida como doble cola, la diferencia que tiene este tipo de cola con Queue es que en una Deque los elementos pueden entrar o salir de los dos extremos de la cola. Este tipo de cola puede usarse para simular el comportamiento FIFO, pero también para simular el comportamiento LIFO (last in first out) en el que el último en entrar es el primero en salir. Este comportamiento suele recibir el nombre de stack o pila, imagina un montón de libros, si en ese montón pones un libro nuevo (último en entrar) cuando retiras un libro será el que esté arriba del todo (último en salir).



Algunas de las **ventajas** del uso de listas son:

- Muy rápido para acceder al primer o último elemento.
- Permite crear colas de elementos muy eficientes.

Algunas de las **desventajas** del uso de listas son:

- Acceso muy lento a los elementos intermedios.

Diagrama para determinar la mejor colección en nuestro programa
El siguiente diagrama puede usarse para tomar decisiones de selección de colección cuando aún no tenemos mucho conocimiento sobre estas y los casos de uso principales de cada una de ellas.

Enlace diagrama

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea un pokédex (solo de 10 pokemon, no hace falta que sean todos). Luego genera 20 capturar aleatorias de pokemons, al final del programa se debe mostrar el estado de la pokédex.

Ejercicio 2.
Crea un pequeño diccionario inglés-español con algunas palabras. El usuario podrá poner una palabra en inglés por consola y el programa responderá con la palabra en español si está en la lista o con un mensaje diciendo que esa palabra no se encuentra en el diccionario.

Ejercicio 3.
Crea una cola de supermercado. Aleatoriamente (70% probabilidades un cliente nuevo se pone en la cola) 30% un cliente de la cola ha sido atendido. Si la cola llega a tener 10 personas suena un mensaje por megafonía diciendo que, por favor, Pepito Pérez vuelva a su puesto de trabajo en Caja. Que se vea por consola todo el proceso (cada vez que alguien entra o sale de la cola y el mensaje por megafonía).

Referencias y ampliación

Colecciones List. Iteradores.

Contenido

Dentro de las colecciones de Java, las que implementan la interfaz List son:

- **ArrayList**: Que usaremos cuando necesitemos acceder a los elementos de forma eficiente por su índice o cuando necesitamos agregar o eliminar elementos al final de la lista de forma frecuente.
- **LinkedList**: Cuando se requieren inserciones o eliminaciones en posiciones intermedias de la lista de forma frecuente y no necesitamos el acceso por índice.
- **Vector**: Es similar a ArrayList pero se usa en escenarios de sincronización. Está deprecado en la actualidad y los escenarios de actualización no están en el alcance de este curso.
- **Stack**: Cuando necesitamos hacer estructuras de datos tipo LIFO.

Analicemos algunos de los métodos de las colecciones para entender sus implicaciones y consecuencias, además esto nos permitirá entender bien los casos de uso.

Si creas un ArrayList y compruebas en el IDE qué métodos puedes usar, verás que algunos son propios de ArrayList (o de sus interfaces dentro del framework de colecciones) y tenemos los métodos típicos heredados de la clase Object.

En este punto de la unidad es importante que entendamos la interfaz de la que parten las colecciones: Iterable.

Iterator
Un iterador (de la interfaz Iterator), es un tipo de objeto que se puede usar para recorrer una a una todas las instancias contenidas en una colección. Estos elementos también reciben el nombre de iteradores universales o cursores.

Para declarar un iterador lo hacemos de la siguiente forma:

```
Iterator<Integer> it = al.iterator();
```

Donde al es una instancia de ArrayList<Integer>.

Iterator es una colección, que a su vez extiende de la colección Iterable. Esta colección tiene un único método “iterator()”, que proporciona una herramienta para recorrer los elementos de una colección de uno en uno.

Cuando tenemos una instancia que extiende de Iterator, en este caso “it”, podremos usar tres métodos sobre ella:

hasNext() - que nos dice si quedan elementos en la colección (con true/false)
next() - que nos retorna el siguiente elemento del iterador.
remove() - elimina el elemento generado tras el último next, si no se usa tras un next generará una IllegalStateException.

Es fácil pensar que todas las operaciones que podemos hacer con un iterador las podemos hacer con un bucle for each, pero no es cierto, ya que no podemos eliminar elementos de una colección mientras la estamos recorriendo con un for each. Además, existen más casos de uso interesantes de iteradores.

Código de ejemplo

Ejercicios

Ejercicio 1.
Existen más tipos de iteradores en Java (Enumeration, ListIterator y Spliterator). Busca información sobre ellos y entiende en qué escenarios se pueden usar cada uno.

Ejercicio 2.
Crea un programa que genere 20 números aleatorios del 1 al 1000, muéstrales por pantalla. Con un iterador muestra todos los números impares. Con otro iterador elimina todos los números múltiplos de tres, diciendo por pantalla qué números han sido eliminados y, finalmente, muestra la lista de números resultantes.

Ejercicio 3.

Crea una lista de 30 palabras de forma aleatoria. Cada palabra se formará por tres letras del abecedario al azar. Muestra la lista por pantalla. Usando un iterador, recorre una segunda vez la lista y las palabras que contengan alguna vocal deben convertirse en mayúsculas. Muestra la lista tras la conversión.
Ejemplo:
rfg
hrw
GHU
ACE
hjl

Referencias y ampliación

Colecciones Set. Comparadores.

Contenido

Dentro de las colecciones de Java, las que implementan la interfaz Set son:

- **HashSet**: Que usaremos cuando necesitemos una lista sin orden y sin duplicados.
- **LinkedHashSet**: Igual que HashSet pero conserva el órden en el que han sido establecidos.
- **TreeSet**: Similar a un HashSet pero ordena automáticamente la lista según un criterio. Al seguir una estructura de árbol es muy rápida para almacenar grandes cantidades de datos que deban ser ordenados.

Las colecciones de tipo Set, como **HashSet**, usan los métodos equals y hashCode de los objetos que almacenan, estos métodos se heredan de la clase Object y tienen mucha importancia en el trabajo con colecciones.

Método equals
El método equals lo hemos usado durante el curso para comparar objetos. Con el uso y la práctica hemos entendido que si queremos comparar dos números enteros lo haremos de la siguiente forma:

2 == 3
Esta expresión nos devolverá un valor booleano (false en este caso), y lo podremos usar para evaluar condiciones u otras operaciones.
También hemos aprendido a comparar cadenas de caracteres, pero en este caso usando equals:
String c1 = “Hola”;
String c2 = “Hola”;
c1.equals(c2);
Esta expresión, al igual que el comparador == nos devuelve un valor booleano (en este caso true). La diferencia entre el primer operador y el segundo es que uno es un tipo primitivo del lenguaje y el segundo (String) es un objeto. Así, equals se emplea para comparar objetos.

Pero, ¿Cómo sabe Java que dos objetos son iguales?
El caso de los String parece sencillo, si pone lo mismo carácter a carácter son iguales, pero ¿para objetos más complejos? Por ejemplo un jugador de baloncesto de la clase Jugador. ¿Cuando es igual a otro?
Está claro que el jugador Pepe Pérez del Avellaneda con dorsal 5 será diferente al jugador Juancho Juanchez del Arcipreste con dorsal 8.
¿Pero este caso?
Pepe Pérez del Avellaneda con dorsal 5.
Pepe Pérez del Arcipreste con dorsal 5.
Tal vez el jugador cambió de club. Es la misma persona, pero ¿es el mismo jugador? Esto Java no lo decide por sí mismo, es el diseñador del programa el que podrá establecer si dos objetos son diferentes o no. ¿Cómo podemos hacer esto? De la misma forma que lo hacíamos con el método toString heredado de Object, si queremos que el método equals tenga un comportamiento concreto debemos sobreescribir el método.

Método hashCode
El método hashCode, heredado de Object, permite conocer una referencia de la instancia de un objeto en memoria. Esta referencia puede ser igual o diferente para dos instancias.

El método hashCode, es un método que usan de forma implícita algunas colecciones de Java, esto implica, que el método se usa en diferentes operaciones con colecciones, tanto si nosotros lo llamamos directamente como si no. Por ejemplo, las colecciones Set, que no permiten duplicados, van a entender que un duplicado es algo

que tenga el mismo hashCode. Puedes profundizar más sobre las funciones hash en el ámbito de las ciencias de la computación en este enlace.

Este método pretende ser una aproximación rápida a la comparación de objetos, no necesariamente precisa en el 100% de los casos, pero sí muy rápida y eficiente, y que asegure sí que son diferentes los códigos hash, los objetos, lo sean, si los código hash son iguales, podría ser que los objetos fuesen iguales o no.

Lo normal es usar los métodos hash generados por defecto por parte del IDE (ver código de ejemplo).

La colección **TreeSet**, es una colección que ordena por defecto. Vamos a ver cómo lo hace mediante los comparadores.

Comparable
Si usamos colecciones ordenadas, como el caso de TreeSet, sus elementos se ordenarán automáticamente según se introduzcan. Por ejemplo si tenemos una colección de Integer estos se ordenarán de forma ascendente, 1, 2, 3, etc. O si tenemos una colección de String se ordenarán alfabéticamente Ana, Beatriz, Carolina, etc. Pero ¿cómo se ordenan los elementos de una clase Alumno, o de una clase Coche? En estos casos Java no sabe qué implica ordenar estas clases definidas por el desarrollador.

Es aquí donde entra la interfaz Comparable, esta interfaz proporciona el método “int compareTo(T objeto)”, que permite comparar dos instancias de objetos de la misma clase y decidir cuál es su orden relativo.

Para este ejemplo de llamada i1.compareTo(i2). El método devolverá un entero con el siguiente significado:

- Si el número retornado es mayor que 0, i1 se ordenará antes que i2.
- Si el número retornado es menor que 0, i2 se ordenará antes que i1.
- Si el número retornado es 0, tienen la misma prioridad de ordenación, a efectos de ordenación son equivalentes.

Ten en cuenta que este método, al igual que pasaba con los métodos equals o hashCode, debemos implementarlo, pero no necesitamos usarlo nosotros en el programa, por ejemplo, una colección TreeSet, lo

usará por defecto para ordenarse cada vez que añadimos un nuevo elemento al Set.

Si una colección, no tiene mecanismos de ordenación automática, como ArrayList, podemos forzar su ordenación mediante el método estático sort de Collections, de la siguiente forma:
Collections.sort(miColeccion);

¿Cómo puedo hacer comparaciones de una clase que yo no puedo modificar?
Para hacer comparaciones es muy sencillo hacer que nuestras clases implementen la interfaz Comparable, pero si la clase no es mía, es de una librería externa o no puedo modificarla por otras razones, no podremos usar Comparable.

Para ello crearemos una clase auxiliar que implemente Comparator y sobreescriba el método compare, como en este ejemplo:

```
class ComparadorCoches implements Comparator<Coche>{
      @Override
      public int compare(Coche a, Coche b) {
      return a.getMatricula().compareTo(b.getMatricula());
      }
}
```

En nuestro programa, llamaremos a Collections.sort con su segundo constructor, que recibe una colección y una clase que implementa comparator:
Collections.sort(comparaCoches, new ComparadorCoches());

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una clase nueva, Jugador, Alumno, FiguraGeometrica, Pokemon, lo que quieras, y crea algunos atributos que tengan sentido. Prueba las opciones de generación automática de equals y hashCode de tu IDE y valora el código generado.

Ejercicio 2.
En el ejemplo proporcionado de secadores, añade el siguiente elemento a la colección misSecadores:
misSecadores.add(new SecadorPelo("Ufesa", "Essential Red", 1900, 21.99));
¿Qué ocurre? ¿Por qué?

Ejercicio 3.
Modifica la clase SecadorPelo, para que en caso de tener la misma potencia dos secadores, se ordene primero el más barato y luego el más caro, y si siguen siendo iguales, se ordenará alfabéticamente por modelo.

Ejercicio 4.
Realiza una colección de la clase que has creado en el ejercicio 1. Asegúrate de que esa clase no implemente la interfaz comparable. Ordena los elementos de la colección mediante un comparator.

Referencias y ampliación

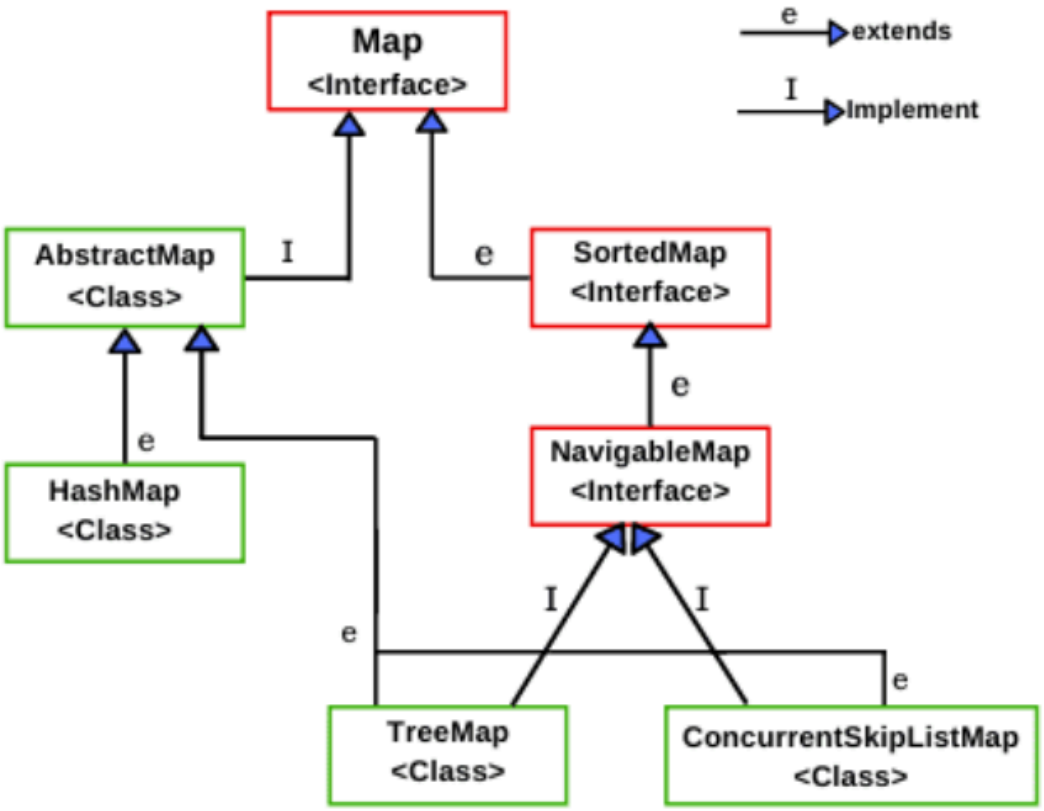
Colecciones Map y Queue/Deque.

Contenido

Map
En el framework de colecciones, las colecciones de tipo Map son aquellas que implementan esta interfaz, y su principal característica es que se organizan en pares clave-valor. Es importante recordar que los elementos de las colecciones siempre son objetos, no tipos primitivos del lenguaje, y esto aplica también a la las claves y los valores de un mapa.

Las **claves deben ser únicas** y estarán asociadas a un valor. Su principio de funcionamiento es muy similar a un diccionario.

En la siguiente imagen podemos ver una jerarquía simplificada de interfaces y clases relacionadas con Map:



- **SortedMap** es un mapa ordenado, similar al principio de funcionamiento que veíamos en la lección anterior con TreeSet.
- **NavigableMap** es un tipo de mapa ordenado, al heredar de SortedMap, que añade funcionalidad adicional para navegar por el mapa entre las ramas y los nodos del mismo.
- **AbstractMap** es una clase abstracta de la que heredan todas las demás clases de mapa.
- **HashMap** usa una tabla hash para almacenar los elementos de la tabla. Es uno de los mapas más usados, para localizar valores, insertar, borrar, etc.
- **TreeMap** es la clase que implementa la funcionalidad de claves-valor de mapas y de estructura de árbol.
- Existen otros mapas como: LinkedHashMap, WeakHashMap o IdentityHashMap que tienen casos de uso más específicos que no están en el alcance de este curso.

Queue y Deque

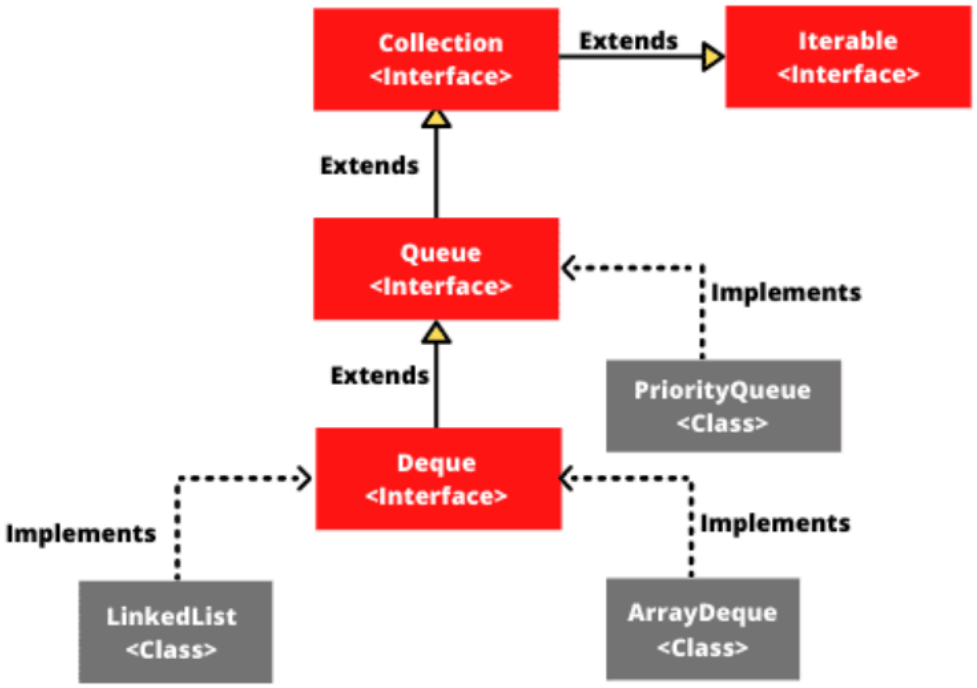
Estas interfaces proporcionan clases con funcionalidad para trabajar con colas. La gestión de colas es un problema habitual en múltiples escenarios de programación y disponer de clases que nos permitan resolver estos problemas de forma sencilla es muy conveniente.

El funcionamiento por defecto de una cola (**Queue**) es FIFO (first in first out), o de forma más sencilla, una cola en una caja de supermercado, donde el primer cliente que se pone a la cola es el primero que será atendido. Sus principales características son:

- Comportamiento FIFO.
- Los elementos sólo pueden obtenerse del inicio de la cola. Método push.
- Los elementos sólo pueden añadirse al final de la cola. Método pop.
- No permite introducir elementos null.

Por su parte **Deque** es una cola en la que podemos trabajar con el inicio y el final de la misma, su nombre proviene de double ended queue. Sus principales características son:

- Comportamiento FIFO o LIFO.
- Los elementos se pueden añadir y quitar del principio y del final de la cola. Métodos addFirst, addLast, removeFirst, removeLast, entre otros.
- No permite introducir elementos null.



Código de ejemplo

Ejercicios

Ejercicio 1.
Crea un programa con un hashMap e introduce algunos datos. Cuando el usuario quiera introducir un nuevo elemento el programa debe informar si ese elemento ya existe y comunicárselo al usuario, si no existe debe añadirlo y comunicar al usuario que el elemento ha sido añadido.

Ejercicio 2.
Escribe un programa que simule el comportamiento de una impresora. La impresora tiene una cola de impresión que almacenará documentos a imprimir. Si la cola llega a 20 elementos no se podrán añadir más documentos y debe generar una excepción.

Referencias y ampliación

UT10 - Lectura y escritura de información

Ficheros 1. Archivos en Java. Clase File.

Contenido

Uno de los principales problemas que hemos encontrado en nuestro aprendizaje del desarrollo de software con Java, es que todos los programas que hemos realizado hasta el momento pierden su información al terminar de ejecutarse.

La gestión de datos e información por parte de los programas informáticos es un componente esencial para poder desarrollar aplicaciones versátiles. En la actualidad existen diferentes métodos de almacenar y obtener datos, algunos de estos son ficheros, bases de datos, consultas a APIs, etc. En esta unidad vamos a entender cómo trabaja Java con los ficheros y algunas de sus utilidades.

Java tiene dos librerías principales para trabajar con ficheros:

- **java.io**. Esta fue la primera librería que se desarrolló y una de las más empleadas.
- **java.nio**. Esta librería se creó para solucionar algunas de las limitaciones de java.io, por ejemplo, java.nio tiene mejores utilidades para realizar operaciones de entrada y salida de alta velocidad.

En esta unidad de trabajo nos centraremos en java.io, ya que una gran cantidad de aplicaciones hacen uso de esta y todos los conocimientos que aprendamos sobre java.io se pueden reutilizar para usar java.nio. El paquete java.io tiene una filosofía de funcionamiento compartida con muchas librerías de trabajo con ficheros en otros lenguajes de programación, entender su funcionamiento nos da una visión global del trabajo con ficheros en el desarrollo del software.

Los ficheros son secuencias de datos almacenados en un dispositivo de almacenamiento, existen diferentes sistemas de gestión de ficheros, pero estos quedan fuera del alcance de este módulo.

Java permite leer y escribir de ficheros de texto (secuencias de caracteres) o ficheros binarios (secuencia de bytes que pueden tener cualquier tipo de información: imágenes, audio, etc.).

En esta lección vamos a entender la clase File. La clase File proporciona múltiples herramientas para trabajar con archivos y directorios. Esta clase no lee y escribe directamente en los archivos, estas operaciones las realizan otras clases que estudiaremos en las próximas lecciones, pero cualquier operación con directorios o ficheros necesitará usar instancias de objetos File.

La clase File, tiene información sobre ficheros y directorios que podremos usar de múltiples formas mediante sus métodos y sus constructores. Explora la información de la API y el código proporcionado para familiarizarte con las herramientas que proporciona la clase File.

Ejemplo de instancia de File:

```
File fichero = new File("miFichero.txt");
```

Cuando trabajamos con ficheros es importante entender las diferentes excepciones que pueden ocurrir en nuestro programa. ¿Qué pasa si intento escribir en un archivo para el que no tengo permisos de

escritura? ¿Y si intento leer un archivo que no existe? El sistema de ficheros es algo ajeno al programa, es por ello que debemos entender todas las operaciones que hagamos con ficheros, investigar si es posible que estas generen excepciones y capturarlas y manejarlas de la forma deseada.

Código de ejemplo

Ejercicios

Ejercicio 1.
Muestra todos los archivos de una carpeta junto con su tamaño. Métodos relevantes: isDirectory, listFiles, isFile, getName, length (en bytes).

Ejercicio 2.
Lista todos los archivos de un árbol de directorios tiene que entrar en tantas carpetas como tenga el árbol (similar a lo que hace el comando tree en linux).

Ejercicio 3.
Explora la clase File en el API de Java e identifica diferentes excepciones generadas por algunos de sus métodos.

Ejercicio 4.
Busca en la API de Java información sobre java.nio y familiarízate con algunas de las diferencias que tiene esta librería sobre java.io.

Referencias y ampliación

Ficheros 2. Lectura y escritura en archivos. Streams.

Contenido

En Java, la entrada y salida de datos se realiza mediante Streams (flujos) de datos. Estos flujos pueden ser conectados a fuentes o destinos, como podrían ser ficheros o conexiones de red. Nosotros vamos a centrarnos en trabajar con ellos sobre ficheros.

Estos flujos parten de dos clases abstractas:

- InputStream, para la entrada de datos. Algunas clases que heredan de esta clase abstracta son:
    - FileInputStream, para leer datos de un archivo.
    - BufferedInputStream, para leer datos usando un buffer, que permite crear almacenamiento de datos intermedio para evitar múltiples llamadas al sistema operativo de lectura, esto mejora la eficiencia leyendo contenidos secuenciales de un fichero.
    - DataInputStream, que permite leer datos primitivos del lenguaje Java directamente desde el fichero.
    - ObjectInputStream, para leer objetos previamente serializados (más adelante en la unidad de trabajo se trata el concepto de serialización).
- OutputStream, para la salida de datos. También tiene clases que heredan de ella con funcionalidad similar a las mencionadas para InputStream, pero con operaciones de salida.

Por otra parte tenemos las clases que nos permiten trabajar con streams (flujos) de caracteres, que están diseñadas para trabajar con textos y realizan automáticamente las conversiones entre bytes y caracteres. Estas clases abstractas son Reader y Writer. Cuando resolvamos problemas que impliquen ficheros de texto, estas clases son más convenientes, pero siempre es conveniente entender la naturaleza de cada clase y los tipos de problemas que resuelven para decidir qué recurso es el mejor en cada ocasión, por ello es importante saber consultar la API.

Algunas clases interesantes que heredan de las clases abstractas Reader y Writer son:

- FileReader/FileWriter: para leer o escribir caracteres de un archivo.
- BufferedReader/BufferedWriter: similar al comportamiento de BufferedInputStream/BufferedOutputStream.
- InputStreamReader/OutputStreamWriter: Estas clases permiten convertir InputStream/OutputStream en clases Reader y Writer.

Cuando trabajamos con ficheros debemos seguir los siguientes pasos:

- Crear un objeto File.

- Abrir el fichero.
- Leer o escribir datos.
- Cerrar el fichero.
- Además, debemos gestionar las posibles excepciones que puedan generar las instrucciones empleadas.

Existen otras clases en Java para trabajar con ficheros PrintWriter y Scanner. Estas clases nos proporcionan más funcionalidades para leer y escribir con ficheros. La clase PrintWriter permite imprimir objetos y datos formateados, por su parte la clase Scanner, es la misma que usábamos al principio de curso para leer datos del teclado, pero la podemos inicializar con un fichero en lugar de indicando la entrada estándar.

Código de ejemplo

Ejercicios

Ejercicio 1.
Investiga cómo puedes hacer que cada vez que se ejecute el fragmento de código que genera el archivo numeros.txt, en lugar de borrar el contenido anterior, se añaden 10 números nuevos sin borrar los anteriores.

Ejercicio 2.
Crea un programa que escriba 10 número aleatorios en datos1.txt, y luego otros 10 números aleatorios en datos2.txt, finalmente debe escribir en datosSumados.txt la suma de cada una de las líneas de datos1 y datos 2. No debes almacenar los números aleatorios en ninguna variable, debes leerlos de los ficheros para realizar la suma. Ejemplo de ejecución:
datos1.txt:
1
2
4

datos2.txt:
4
6
7

datosSumados.txt:
5
8
11

Ejercicio 3.
Crea un programa que escriba cinco números aleatorios del 1 al 9 en cada línea, luego lea todos los números del archivo y añada a cada línea el valor de los números sumados hasta dicha línea. No debes crear variables intermedias que guarden los números, debes escribir el archivo y después leer y escribir el archivo de nuevo. Ejemplo de ejecución, contenido del archivo:
3 (suma:3)
5 (suma:8)
2 (suma:10)
7 (suma:17)
1 (suma:18)

Referencias y ampliación

Ficheros 3. Buffers.

Contenido

¿Qué son los buffers?
Los buffers son herramientas ampliamente empleadas en la electrónica digital o en la computación. Un buffer es un dispositivo o región de memoria que permite el almacenamiento temporal de datos dentro de un sistema digital.

La mayoría de las funciones de los buffers son transparentes para los usuarios de sistemas digitales, pero permiten optimizar rendimientos u ofrecen una mejor experiencia de usuario.

Vamos a ilustrar el uso de un buffer con algo que es familiar a todo el mundo, el consumo de contenido multimedia, por ejemplo, viendo un vídeo de Youtube, este debe llegar a tu equipo desde los servidores y mostrarse en tu equipo a una velocidad fija.

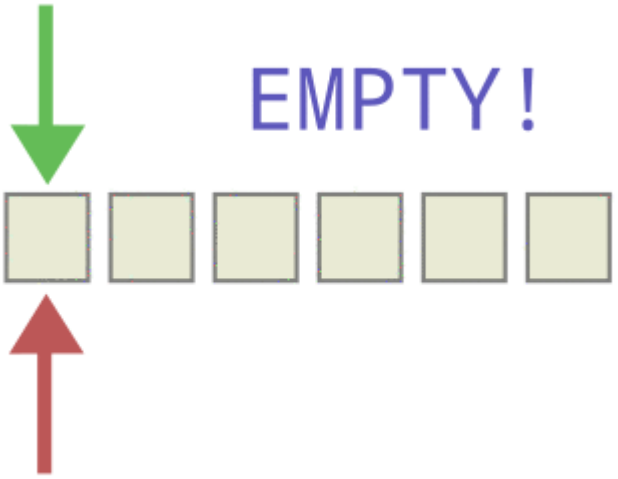
En este escenario debemos obtener datos de una fuente y mostrar dichos datos en el destino (herramienta de reproducción del vídeo para que se emita por la pantalla y los altavoces). En este escenario imaginemos que el vídeo ocupa 100MB y que la calidad del vídeo es de 5MB por segundo, siendo un vídeo de 20 segundos de duración en el que la información de cada segundo son 5MB.

Si quiero reproducir el vídeo a su velocidad natural, necesitaré que me llegue esta información con una cadencia exacta de 5MB por segundo, para que mi reproductor tenga esa información en el momento exacto. Pero el vídeo no se reproduce segundo a segundo, si no frame a frame, o muestra de audio a muestra de audio. En este escenario, se requeriría una sincronización óptima y una latencia nula para poder recibir la información de los servidores y mostrarla.

Este escenario no es realista, por ello se usa un buffer para gestionar esa información. Nuestro dispositivo solicitará la información del vídeo al servidor, y este se lo mandará por bloques, este envío tendrá incertidumbres basadas en la cantidad de peticiones a los servidores, el ancho de banda de la conexión, la congestión en diferentes tramos de las redes por las que pase la información, etc. Si mi dispositivo tiene un buffer en el que puede almacenar la información que le llegue, y puede ir consumiendo dicha información para mostrar el vídeo, existirán dos procesos de entrada y salida de información que no tendrán una gran sensibilidad, el buffer se irá llenando en función de las capacidades de servicio de la información, y el dispositivo irá consumiendo dicha información a la cadencia necesaria para mostrar el vídeo de forma homogénea y satisfactoria para el usuario.

Este tipo de comportamiento se replica en muchos escenarios, sobre todo en aquellos donde el tiempo de las operaciones no sea determinista. Entender este concepto es importante para entender la utilidad de los buffers. El estudio en profundidad de este mecanismo está fuera del alcance del módulo, pero tener una concepción del problema que resuelven es importante para entender su uso.

Ejemplo gráfico de funcionamiento de un buffer:



En Java existen clases abstractas e interfaces que permiten realizar estructuras de datos para emular este comportamiento, pero están fuera del alcance de este curso. Nos centraremos en trabajar con dos clases específicas que permiten leer y escribir en ficheros de forma optimizada gracias a buffers.

BufferedWriter y BufferedReader
Estas dos clases (BufferedWriter y BufferedReader) pertenecen al paquete java.io y proporcionan un buffer de almacenamiento para escritura y lectura, con el objetivo de mejorar el rendimiento de las operaciones de entrada y salida, ya que minimizan los accesos a recursos del sistema, al leer bloques de información más grandes que luego se consumen al ritmo necesitado por las aplicaciones.

Código de ejemplo

Ejercicios

Ejercicio 1.
Enlace a El Quijote de Cervantes en Gutenberg library:
https://www.gutenberg.org/cache/epub/2000/pg2000.txt

Realiza una lista de las 100 palabras más repetidas en el libro. Para contar las palabras usa la colección o colecciones óptimas y optimiza la lectura del fichero.
Nota: Puedes suponer que las palabras junto a signos de puntuación o acentos son diferentes, por ejemplo, que y ¿que pueden ser palabras diferentes, como y cómo pueden ser diferentes.

Referencias y ampliación

Ficheros 4. Archivos binarios.

Contenido

Hasta ahora hemos trabajado con ficheros de texto, esto nos ha sido muy conveniente para poder ver los resultados de nuestros programas y por que es muy común trabajar con estos ficheros para generación de informes, logs, archivos de configuración, etc.

En esta lección trabajaremos con ficheros binarios, a estos ficheros también es frecuente llamarlos ficheros de datos, y están formados por secuencias de bytes. Estos ficheros suelen ser más reducidos en tamaño, ya que guardan en ellos la información mínima necesaria, ya que si queremos guardar un entero (int) necesitaremos 4 bytes o para un long 8 bytes, sin necesidad de tener en cuenta una decodificación (formato). Por otra parte, las personas no podremos leer la información de estos ficheros directamente.

Muchas aplicaciones serán más eficientes guardando la información en ficheros binarios, sobre todo, cuando el usuario no necesite visualizar en ningún momento estos archivos de forma directa. Algunos ejemplos de uso de archivos binarios podrían ser: almacenamiento de datos estructurados, persistencia de datos, archivos multimedia o archivos de sistema, entre otros.

En esta lección no vamos a profundizar en los casos de uso de ficheros binarios, tan solo haremos algún ejemplo para entender cómo leer y escribir información es este tipo de ficheros.

En esta lección nos centraremos en almacenar en nuestros ficheros binarios información de los tipos primitivos del lenguaje Java, como son boolean, char, byte, int, short, long, float y double.

Las operaciones de lectura y escritura son muy similares a las que hemos trabajado con ficheros de texto. Las clases principales involucradas en la lectura y escritura de archivos binarios son DataInputStream y DataOutputStream.

Estas clases permiten leer y escribir diferentes tipos primitivos del lenguajes con métodos como: writeBoolean, WriteInt, readBoolean o readInt, entre otros. Consulta la información de la API para saber más sobre sus constructores y métodos.

Para poder leer los archivos de forma correcta debemos saber cómo se han escrito dichos archivos, ya que nosotros, al leer no podremos saber si lo que estamos leyendo era un int o un float en el momento que se almacenó.

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea un archivo binario con la información de 50 Pokémon, de forma aleatoria. Ten en cuenta que la información que debes almacenar es: Número de Pokémon (ten en cuenta que en la actualidad hay 1025 Pokemon, desde 1-Bulbasaur hasta 1025-Pecharunt), el género, y las siguientes estadísticas (hp, ataque, defensa, ataque especial, defensa especial y velocidad).
Teniendo en cuenta que las estadísticas pueden tomar valores del 20 al 260, usa los tipos de datos que minimicen el tamaño del fichero.
Luego lee el contenido del fichero y vuelta los datos a un fichero en formato texto con la información de la siguiente forma para cada Pokémon:

```
*********************************************
* Número: 123                                *
* Género: Hembra                             *
```

```
* Estadísticas:                                *
*   HP           (054) [----                ] *
*   Ataque       (189) [---------------     ] *
*   Defensa      (085) [-------             ] *
*   Ataque esp.  (154) [------------        ] *
*   Defensa esp. (214) [-----------------   ] *
*   Velocidad    (078) [------              ] *
************************************************
```

Entre corchetes deben aparecer guiones proporcionalmente al valor de las estadística, si el valor fuese 260 tendría los 20 guiones, y si el valor es 20 tendría un guión.

Ejercicio 2.
Crea una programa que permita generar ficheros binarios con mensajes cifrados. Para cifrar los mensajes usaremos el siguiente algoritmo:

- Generamos números (short) pseudoaleatorios mediante la clase Random y una semilla (sobrecarga del constructor de Random).
- Cada char de nuestra cadena de texto sin cifrar se suma al número aleatorio y se guarda en el fichero con formato int.
- Para decodificar se leen los datos como int y se restan los números aleatorios de la semilla, y se transforma en char.

Crea un programa que pueda codificar mensajes y otro que pueda leerlos, pero solo funcionará si ambos conocen la semilla.

Referencias y ampliación

Ficheros 5. Archivos de acceso aleatorio.

Contenido

Hasta ahora hemos accedido a nuestros ficheros de forma secuencial, esto es leyéndolos o escribiendolos desde el principio hasta el final, pero y si solamente necesito leer o escribir en una posición concreta del archivo.

Estos tipos de acceso reciben el nombre de acceso aleatorio (random access) en el ámbito de las ciencias de la computación. En esta lección veremos una clase que nos permite acceder de esta forma a nuestros ficheros (RandomAccessFile).

A diferencia de las clases empleadas en ejemplos anteriores de esta lección, no tenemos dos clases una para leer y otra para escribir. En este caso RandomAccessFile permite realizar las dos operaciones, pero para ellos debemos indicar en su constructor qué tipos de acceso queremos realizar. Es conveniente que veas qué tipos de acceso podemos usar en el constructor consultando la API, pero aquí se ponen dos ejemplos típicos:

```
//solo lectura
RandomAccessFile soloLectura = new RandomAccessFile(“datos.dat”, “r”);
//lectura y escritura
RandomAccessFile lecturaEscritura = new RandomAccessFile(“datos.dat”, “rw”);
```

Si consultas la API verás que esta clase tiene muchos métodos similares a los que empleamos en los accesos a archivos binarios. Pero hay un método especial llamado seek.

El método seek(long position) permite establecer un puntero que dice cual es la posición del fichero sobre la que se ejecutará la siguiente operación. Esta posición se determina en número de bytes respecto del inicio del fichero.

Por ejemplo:

- seek(0): apuntará al primer elemento del fichero.
- En caso de que nuestro fichero tenga 10 números enteros (4 bytes por entero), seek(8) apuntará al tercer entero.

Ten en cuenta que las operaciones de lectura y escritura mueven el puntero, así al leer un entero, no solo obtengo ese entero sino que además el puntero se desplaza 4 bytes.

Código de ejemplo

Ejercicios

Ejercicio 1.
Intenta leer el valor del atributo velocidad del 6º Pokémon del ejercicio 2 de la lección anterior.

Ejercicio 2.
Crea un programa que permita jugar al 3 en raya. El tablero tendrá estas posiciones:
1 2 3
4 5 6
7 8 9
El programa debe aceptar como comandos los números del 1 al 9 y la letra “r”.
Si se introduce un número se ubicará el símbolo correspondiente en la ubicación deseada siempre que sea posible.
Si se introduce “r”, se deshará el movimiento anterior. Esta función podrá usarse hasta reiniciar totalmente el tablero si se quiere, cada pulsación deshace un movimiento.
El programa debe detectar si se ha terminado la partida y anunciar al ganador o tablas.
Se debe emplear un archivo para almacenar los movimientos.

Referencias y ampliación

Ficheros 6. Serialización de objetos.

Contenido

Vale, hasta el momento hemos guardado texto y datos primitivos del lenguaje Java, pero, ¿qué pasa con los objetos? ¿Existe alguna forma de guardar la información de estos? Sí, mediante la serialización.

La serialización nos permite convertir el estado de una instancia de un objeto a datos, podremos guardar estos datos en un fichero para recuperarlos posteriormente (el proceso de obtener los datos y generar con ellos la instancia del objeto se llama deserialización, la serialización también se usa para almacenar los objetos en bases de datos o para transmitirlos por redes).

Para poder serializar objetos estos deben implementar la interfaz Serializable. Si queremos serializar un objeto de la propia API de Java tendremos que asegurarnos que el objeto implementa esta interfaz, y si queremos que un objeto propio pueda ser serializado tendremos que hacer que implemente la interfaz.

La interfaz Serializable no nos obliga a implementar ningún método, cuando hacemos que una clase implemente de Serializable habilitamos que esta pueda ser serializada y deserializada, pero no necesitamos hacer nada más que poner el “implements Serializable” en la definición de la clase.

Si un objeto contiene tipos primitivos del lenguaje como atributos podrá ser serializada, pero si el objeto contiene instancias de otros objetos, solo podrá ser serializada si dichos objetos implementan la interfaz Serializable a su vez.

Para escribir el objeto serializado emplearemos la clase ObjectOutputStream y su método writeObject.

Para leer el objeto serializado emplearemos la clase ObjectInputStream y su método readObject, este método retornará un objeto de tipo Object, con lo cual tendremos que hacer un cast del objeto a nuestro tipo de objeto deseado.

Cuando trabajamos con objetos serializados debemos tener en consideración un nuevo tipo de excepción: ClassNotFoundException. Esta excepción puede ocurrir al tratar de cargar una clase serializada, pero esta no se encuentra entre los recursos accesibles por nuestro programa.

Código de ejemplo

Ejercicios

Ejercicio 1.
Añade un nuevo comando al ejercicio 2 de la lección anterior (tres en raya), este será el comando “s”. Al pulsar este comando se guardará el estado de la partida. Cuando el programa se inicie deberá cargar automáticamente el estado de la partida si existe un archivo guardado. Al terminar la partida por victoria o tablas no debe haber ningún archivo de guardado.

Ejercicio 2.
Modifica la práctica del lienzo para que al salir se guarde el estado del lienzo y al entrar se cargue el estado del mismo. Crea una nueva opción para resetear el lienzo.

Referencias y ampliación

UT11 - Interfaces gráficas

Interfaces gráficas 1. Interfaces gráficas en Java. Contenedores.

Contenido

¿Qué es una interfaz gráfica?
Las interfaces gráficas de usuario, también conocidas como GUI, son aplicaciones software que permiten al usuario interactuar con ellas mediante elementos gráficos para representar información y reaccionar a las interacciones del usuario.

El propósito por el que se desarrollan las interfaces gráficas es para dar acceso a los recursos del sistema a usuarios sin necesidad de conocimientos específicos del mismo. Los sistemas operativos modernos, la gran mayoría de aplicaciones o todo el ecosistema web están basados en interacción mediante interfaces gráficas.

En esta unidad de trabajo no profundizaremos sobre principios de diseño o detalles de bajo nivel sobre el funcionamiento de las interfaces, ya que quedan fuera del alcance de este curso. En la unidad de trabajo entenderemos el funcionamiento de algunas de las interfaces gráficas de Java y haremos ejemplos sencillos de programas con interfaz gráfica, entendiendo su filosofía de funcionamiento.

Librerías Java para el desarrollo de interfaces gráficas

En la actualidad Java no es uno de los lenguajes más empleados para desarrollar interfaces de usuario, pero dada su evolución histórica el lenguaje cuenta con algunas librerías de desarrollo de interfaces. Entendiendo cómo funcionan estas interfaces, algunos principios de diseño y la filosofía tras los componentes y eventos, podremos aprender a desarrollar interfaces en otros lenguajes de programación o con otras tecnologías de forma sencilla gracias al conocimiento aprendido en esta unidad de trabajo.

Java cuenta con tres librerías para el desarrollo de interfaces de usuario:

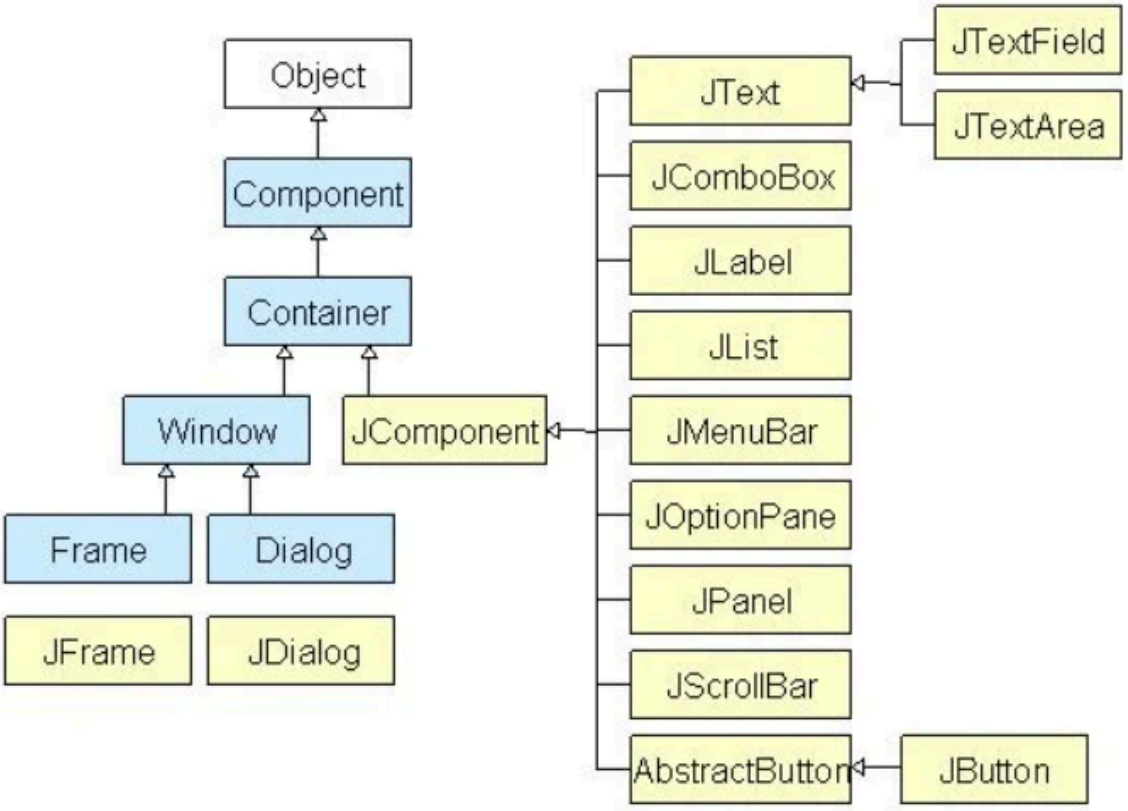
- AWT. Primera librería gráfica, que empleaba los componentes gráficos del propio sistema operativo en el que se ejecuta.
- Swing. La segunda librería gráfica fue Swing, que implementa nativamente en Java los componentes y traía algunas mejoras sobre AWT. Cuando se desarrollan interfaces gráficas con Swing se siguen empleando algunas clases de AWT, puesto que siguen siendo válidas.
- JavaFX. Con la evolución de las interfaces gráficas debido a los dispositivos móviles y la web, las herramientas proporcionadas por Swing eran insuficientes y se desarrolló JavaFX para aportar nuevas funciones y un desarrollo de interfaces más fácil de adaptar y de aplicar estilos, similar a otras herramientas modernas de diseño de interfaces.

En esta unidad de trabajo emplearemos Swing y AWT, que nos aportan una aproximación sencilla al desarrollo de interfaces gráficas y sigue teniendo un gran valor didáctico.

Java Swing

Java Swing nos aporta un conjunto de componentes que podremos usar para desarrollar interfaces gráficas, estos componentes pueden ser etiquetas de texto, cajas para introducir texto por parte del usuario, botones, listas, etc.

En la siguiente imagen puedes ver un resumen de la jerarquía de componentes de Swing:



JFrame

El componente principal sobre el que se va a crear la interfaz gráfica en Swing es de la clase JFrame(tutorial de la documentación de Oracle para crear GUI con Swing).

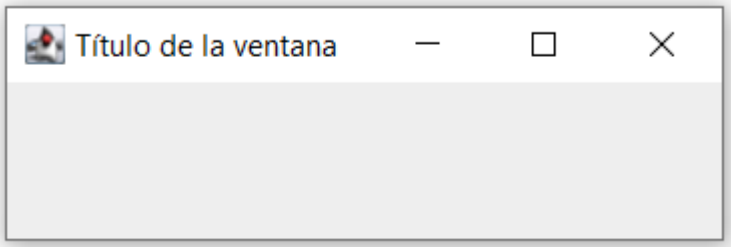
JFrame crea una ventana en nuestro sistema gráfico de ventanas de nuestro sistema operativo. JFrame es solo el contenedor de la ventana, pero debemos añadir paneles u otros elementos para generar la interfaz de usuario.

Otra clase de tipo ventana son los JDialog, que veremos en próximas lecciones de esta unidad de trabajo.

Así podemos crear un JFrame:

```
JFrame miFrame = new JFrame("Título de la ventana");
```

Solo con un JFrame tendremos algo así:



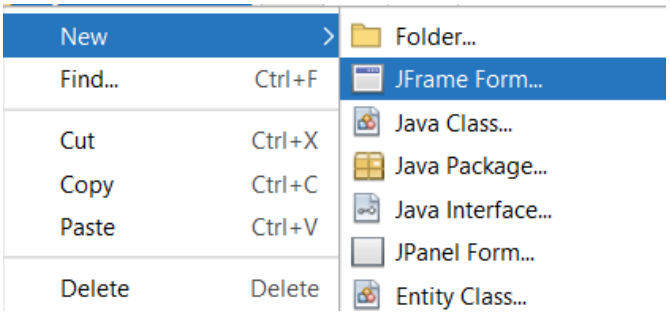
JPanel
JPanel es un contenedor que puede agrupar un conjunto de componentes. Este tipo de componente debe estar contenido dentro de un JFrame.

Así podemos crear un JPanel y añadirlo al JFrame:

```
JPanel panel1 = new JPanel();
miFrame.add(panel1);
```

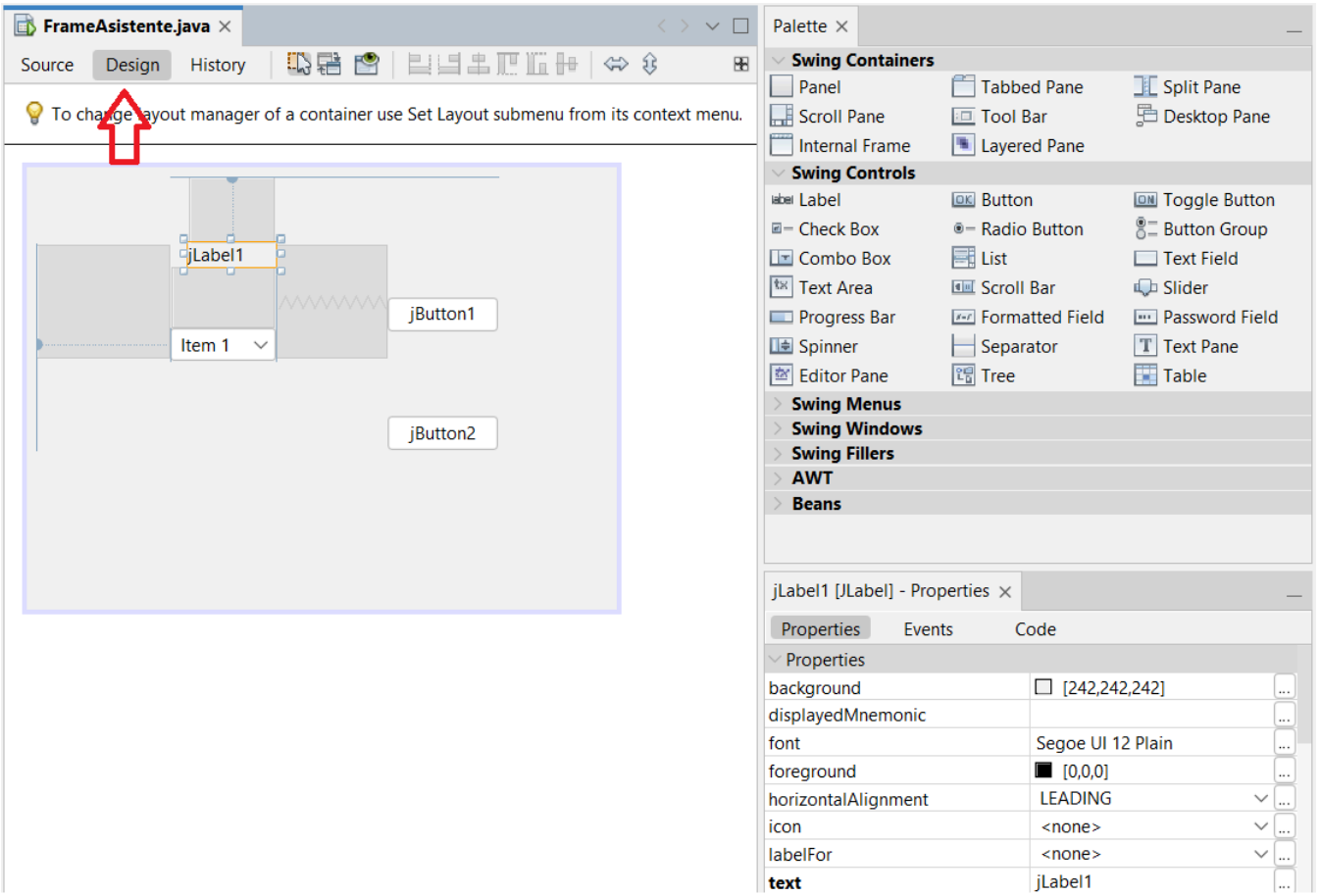
Asistentes de diseño de interfaces
Algunos IDE de desarrollo de software ofrecen herramientas de asistencia y creación de código automático para ayudar con el desarrollo de interfaces gráficas. Al igual que podemos generar constructores, métodos getters y setters, equals y hashCode de forma automática, las herramientas de diseño gráfico de interfaces de usuario alivian la tarea de creación del código de interfaces. Es importante entender que estas herramientas ayudan con la generación, pero no generan software que cumplan con los requisitos de nuestra aplicación; su labor es crear el código más monótono de forma rápida para que podamos adentrarnos en desarrollar las partes más relevantes y relacionadas con los requisitos propios de nuestra aplicación.

En NetBeans, si generamos una clase gráfica, como JFrame o JPanel:



El IDE nos permitirá ver su código fuente, como con todas las demás clases, pero también ofrecerá un modo “Design” en el que podremos

crear algunas partes de nuestra interfaz gráfica mediante un asistente, en el que podremos arrastrar los componentes y configurar sus propiedades de forma sencilla. Es importante entender el código que generan estas herramientas automatizadas, si tras hacer las configuraciones pinchamos en el modo “Source”, podremos ver todo el código fuente que ha creado NetBeans con nuestra configuración en el modo “Design”.



Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una clase de tipo JFrame con NetBeans. Empleando el diseñador de interfaces prueba diferentes componentes, modifica sus propiedades y parámetros. Finalmente lee el código generado por la herramienta e intenta entender la relación entre lo que has hecho de forma gráfica y el código generado.

Ejercicio 2.
Prueba a modificar las propiedades de “panel1” en el código adjunto con esta lección. Familiarízate con algunos de sus métodos.

Referencias y ampliación

Interfaces gráficas 2. Layouts y Componentes.

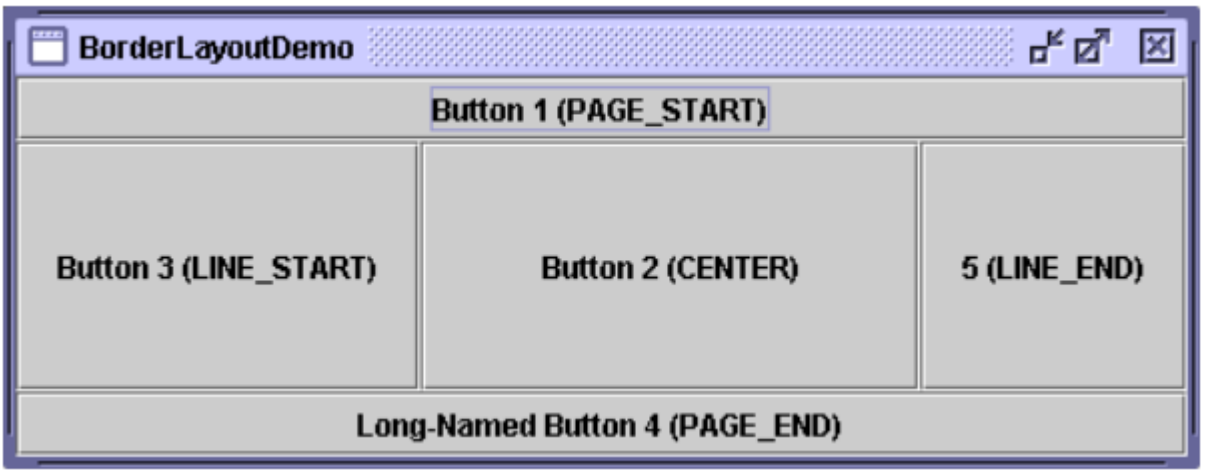
Contenido

Layout managers
Para controlar la disposición de los componentes en nuestras interfaces gráficas, Java Swing hace uso de los gestores de diseños (layout managers).

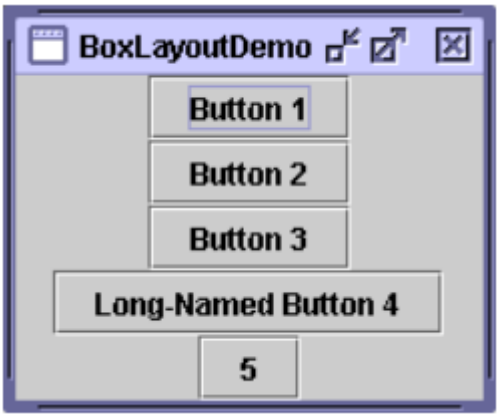
Para emplearlos podemos crear instancias de objetos de tipo layout que incluye Swing y asignarlos a nuestros JFrames o JPanel.

Ejemplo:
```
JFrame miVentana = new JFrame();
BordeLayout miLayout = new BorderLayout();
miVentana.setLayout(miLayout);
```

BorderLayout: Posiciona los componentes en las coordenadas arriba, abajo, izquierda, derecha y centro.



BoxLayout: Posiciona los componentes en una fila o columna. Mantiene el tamaño de los componentes y permite alinearlos.

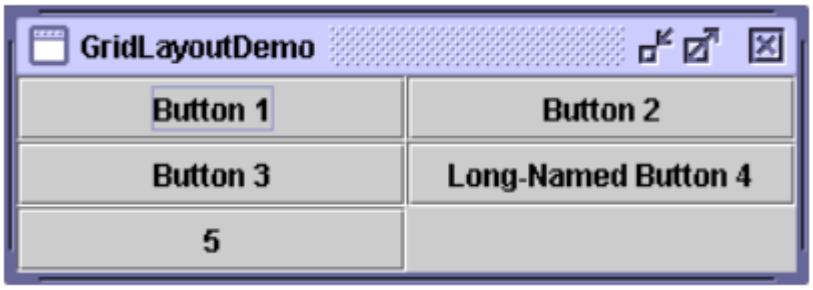


CardLayout: Se emplea para manejar diferentes JPanel que ocupan el mismo espacio. Puedes seleccionar los diferentes paneles mediante un combobox.

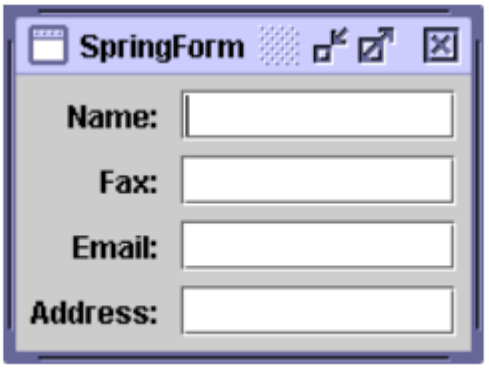
FlowLayout: Es el layout por defecto de JPanel. Los posiciona en fila, si no hay suficiente espacio salta a una nueva fila.

GridBagLayout: Usa una rejilla para posicionar los elementos, cada elemento puede ocupar un tamaño diferente en la rejilla.

GridLayout: Estructura los componentes en una rejilla en la que cada celda es del mismo tamaño.



SpringLayout: Layout más complejo que permite especificar con precisión las relaciones entre cada componente.



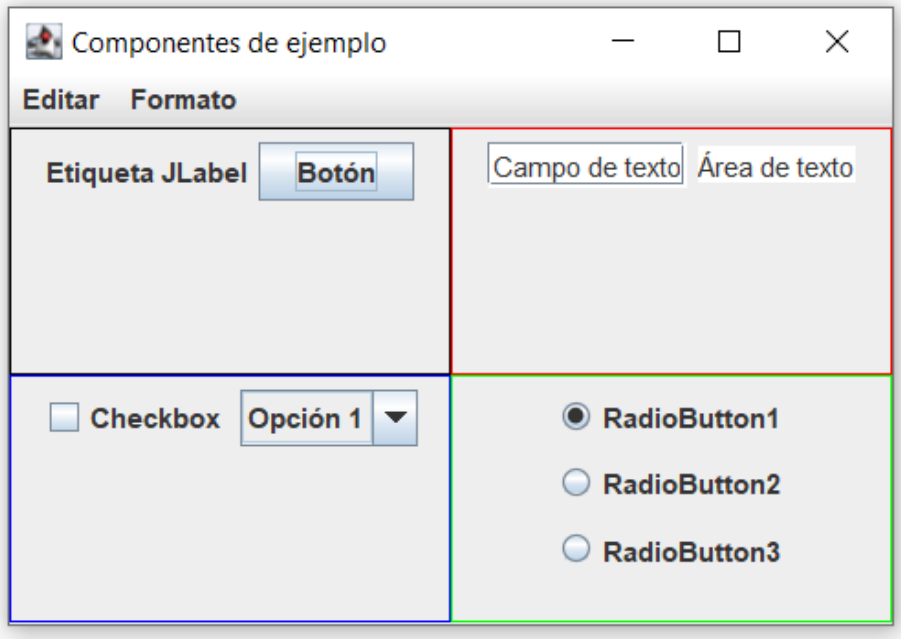
Puedes profundizar más sobre los diferentes tipos de layout y su uso en este enlace o en el propio tutorial de Oracle.

Componentes

Los componentes son los elementos visuales que usamos en las interfaces gráficas. Java Swing tiene clases para implementar muchos de los componentes más habituales en el desarrollo de interfaces gráficas.

En el código de ejemplo de esta lección se encuentran ejemplos de los siguientes componentes (ten en cuenta que existen más componentes en la librería Swing):

- JLabel: permite mostrar un texto.
- JButton: Permite crear botones con los que podrá interactuar el usuario.
- JTextField: Permite crear cajas de texto para que el usuario introduzca datos.
- JTextArea: Similar al anterior pero que permite múltiples líneas de texto.
- JMenuBar: Permite crear barras de menú horizontales, cada elemento de la barra será un JMenu que tendrá dentro JMenuItem.
- JCheckBox: Permite crear casillas de verificación.
- JRadioButton: Permite crear grupos de opciones de selección, en los que solo se podrá seleccionar uno de los elementos del grupo.
- JComboBox: Permite crear una lista desplegable en la que el elemento seleccionado es el elemento visible.



Código de ejemplo

Ejercicios

Ejercicio 1.
Prueba a modificar los diferentes Layouts en el ejemplo proporcionada con los componentes.

Ejercicio 2.
Crea un JFrame usando la herramienta de diseño de NetBeans e inserta todos los componentes gráficos del interfaz. Familiarízate con sus parámetros y características.

Referencias y ampliación

Interfaces gráficas 3. Eventos.

eventos (action listeners y key listeners) eventos ratón mouselistener

Contenido

¿Qué son los eventos?
Los eventos son acciones desencadenadas por la interacción del usuario u otros actores. Estos eventos van a permitir que el programa interactúe con sus peticiones, por ejemplo, si hemos programado una calculadora, esta tendrá que ser capaz de reaccionar a las pulsaciones de los botones por parte del usuario para decidir qué número y operaciones debe usar.

El evento, dentro de nuestro programa, desencadenará la ejecución de una función o parte de nuestro código, y podrá ser generado por diferentes fuentes, como la pulsación de un botón por parte del usuario (pulsar un botón, pulsar un tecla, etc.).

¿Cómo se programan los eventos?
Para programar los eventos haremos uso de interfaces (como la interfaz de AWT ActionListener), estas interfaces proporcionan funcionalidad a nuestras clases para reaccionar a estos eventos.

La interfaz ActionListener nos forzará a implementar el método actionPerformed que será invocado cuando ocurra la acción. Es en este método donde debemos poner el código que queremos ejecutar cuando se desencadene la acción.

¿Cómo indico a un componente de mi interfaz gráfica que genere eventos?
Para que un componente de nuestras interfaces gráficas genere eventos debemos usar su método addActionListener, que no devuelve nada y recibe como parámetro cualquier objeto que implemente la interfaz ActionListener (polimorfismo):

```
boton1.addActionListener(ActionListener l);
```

En este enlace puedes encontrar información sobre los diferentes listeners que pueden implementarse en cada componente.

Eventos de teclado
Como se mostraba en el enlace anterior, además de ActionListener, los elementos de nuestras interfaces gráficas pueden reaccionar a más eventos.

Para leer eventos del teclado usaremos la interfaz KeyListener. Esta interfaz nos obliga a implementar tres métodos abstractos:

- keyPressed(KeyEvent e): se invoca al pulsar una tecla.
- keyReleased(KeyEvent e): se invocará al soltar la tecla.
- keyTyped(KeyEvent e): se invocará cuando la tecla que hemos pulsado genere un carácter Unicode. Esto es útil cuando queremos detectar eventos relacionados con la escritura. No se activará, por ejemplo, con las teclas Alt y Control.

Para añadir un KeyListener en alguno de nuestros componentes usamos addKeyListener, pasando como parámetro la instancia de un objeto que implementa la interfaz KeyListener, de la siguiente forma:

componente.addKeyListener(KeyListener k);

Eventos del ratón
Para leer eventos del ratón usaremos la interfaz MouseListener. Esta interfaz nos obliga a implementar cinco métodos abstractos:

- mouseClicked(MouseEvent e): Al hacer click con el ratón (pulsar y soltar) sobre el componente.
- mousePressed(MouseEvent e): Al pulsar el botón del ratón sobre el componente.
- mouseReleased(MouseEvent e): Al soltar el botón del ratón sobre el componente.
- mouseEntered(MouseEvent e): Cuando el puntero del ratón entra sobre el área del componente.
- mouseExited(MouseEvent e): Cuando el puntero del ratón sale del área del componente.

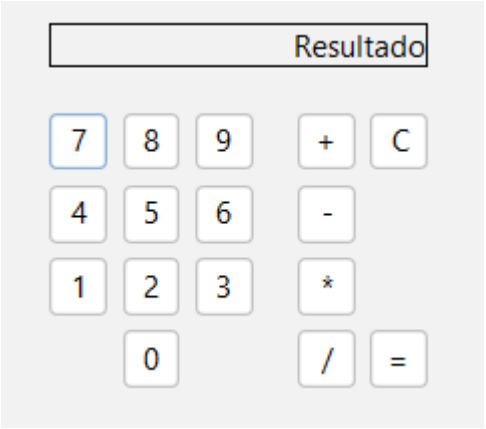
Para añadir un MouseListener en alguno de nuestros componentes usamos addMouseListener, pasando como parámetro la instancia de un objeto que implementa la interfaz MouseListener, de la siguiente forma:

componente.addMouseListener(MouseListener m);

Código de ejemplo

Ejercicios

Ejercicio 1.
Crea una calculadora como la siguiente:



Donde todo son botones, y “Resultado” es un JLabel. El botón C sirve para eliminar toda la información almacenada.

Ejercicio 2.
Prueba la gestión de eventos con otros componentes, como JCheckBox, JRadioButton o JComboBox.

Ejercicio 3.
Experimenta con el jframe4 del código proporcionado, para ver qué teclas se activan con el método keyTyped.

Ejercicio 4.
Usando el código proporcionado con la lección, emplea las herramientas de depuración sobre la clase “ReaccionEvento” para analizar la instancia “e” de “ActionEvent”. ¿Qué métodos o propiedades aporta? ¿Piensa algún uso de estas?

Ejercicio 5.
En la línea 32 del main del programa proporcionado con la lección tenemos esta línea:
boton2.addActionListener(new ActionListener(){
NetBeans genera un warning sobre ella. ¿Qué significa? Intenta hacer la solución que el IDE te propone.
¿No lo tienes claro del todo? Explora los recursos de ampliación de la unidad.

Referencias y ampliación

Interfaces gráficas 4. Ventanas emergentes.

ventanas emergentes(joptionpane jdialog jpopupmenu) jFileChooser JColorChooser

Contenido

Ventanas emergentes
Las ventanas emergentes son elementos gráficos adicionales a nuestra interfaz que tienen como propósito mostrar o recoger información al usuario.

Estas ventanas no alteran el diseño o contenido de nuestra aplicación principal, ya que al ser emergentes, podemos mostrar mensajes adicionales sin necesidad de modificar el diseño de nuestras ventanas principales.

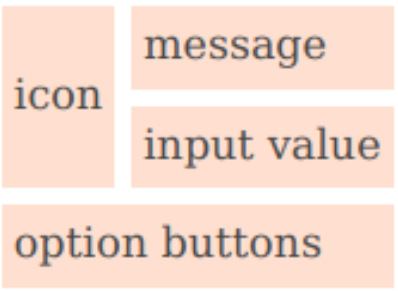
En Java Swing existen diferentes ventanas emergentes.

JOptionPane. Esta clase permite generar una ventana emergente modal (que bloquea el acceso a su ventana origen) con un cuadro de diálogo para informar o requerir información al usuario. Sus principales **métodos son estáticos**, lo que nos permite crear ventanas emergentes sin crear instancias de JOptionPane. Además, estos métodos están sobrecargados para aceptar diferente número o tipo de parámetros.

Sus principales métodos son los siguientes:

- **showConfirmDialog**: Solicita una confirmación al usuario con botones como Si/No/Cancelar.
- **showInputDialog**: Solicita información al usuario.
- **showMessageDialog**: Muestra información al usuario.
- **showOptionDialog** : Permite configurar diferentes opciones.

El esquema gráfico de esta ventana emergente es este:



JDialog. Esta clase permite crear ventanas emergentes personalizadas. Estas ventanas podrán contener cualquier otro tipo de componente y podremos decidir si su comportamiento es modal o no. Cuando necesitemos crear ventanas emergentes y las opciones de JOptionPane no sean suficientes, JDialog será la mejor opción.

JPopupMenu. Es una ventana con un menú contextual (el típico menú que aparece cuando pulsamos botón derecho sobre alguna interfaz gráfica).

Selectores de archivos

En Java Swing existen algunas ventanas adicionales que permiten hacer operaciones habituales en el uso de interfaces gráficas.

Una de estas ventanas es la que nos permite seleccionar un archivo o carpeta de nuestro equipo. Gracias a esta ventana, podremos trabajar con ficheros en nuestras aplicaciones con interfaces gráficas, dando al usuario la opción de seleccionarlos según sus necesidades.

JFileChooser. Esta clase nos permite generar un componente gráfico de navegación sobre el sistema de archivos, para seleccionar archivos o directorios.

Los métodos principales que usaremos son:

- **showOpenDialog**: para abrir archivos.
- **showSaveDialog**: para guardar archivos.

Código de ejemplo

Ejercicios

Ejercicio 1.

Crea una ventana que permita introducir usuario y contraseña, con un botón “Enviar”. Crea una ventana emergente para informar al usuario.

Si los datos son correctos será informativa, si son incorrectos será de error.

Ejercicio 2.
Crea un programa que permita al usuario seleccionar un archivo de texto y muestre en una etiqueta las 50 primeras letras de ese archivo de texto.

Referencias y ampliación

Interfaces gráficas 5. Imágenes. Graphics.

Contenido

preferences?
java.util.prefs

graphics graphics2D

Código de ejemplo

En esta lección vamos a trabajar con algunos elementos gráficos que podemos añadir a nuestras interfaces de usuario Java Swing. Usaremos iconos, imágenes y componentes gráficos. Además, entenderemos algunos métodos especiales que usa Java Swing para redibujar los elementos en pantalla, y veremos cómo usarlos para crear interfaces.

Trabajar con imágenes
ImageIcon. Esta clase implementa la interfaz Icon, que permite pintar iconos a partir de imágenes. Las imágenes pueden enlazarse como archivos, arrays de bytes o URLs, pero nosotros nos centraremos en hacerlo con archivos.

Podremos crear estos iconos en diferentes componentes, como etiquetas, botones o paneles.

Image. Esta clase abstracta es la superclase de la que heredan todas las clases que desarrollan imágenes. Es importante tenerla en

consideración, ya que muchos de los métodos que usaremos con diferentes clases de imágenes vendrán de esta clase abstracta.

BufferedImage. Esta clase hereda de la clase Image y gestiona los datos de la misma con un buffer.

Trabajar con gráficos
Java usa la clase Graphics para trabajar con elementos gráficos. Esta clase tiene métodos para dibujar figuras y textos.

paintComponent y repaint
El método paintComponent es llamado por Java cuando el programa con interfaz gráfica Swing detecta que debe dibujar el componente, cuando se crea por primera vez, cuando se redimensiona la ventana o cuando el código llama al método repaint, que forzará a ejecutar de nuevo el código de paintComponent.

Nosotros podemos sobrescribir este método para una clase propia que herede de algún elemento gráfico, como JPanel.

Timer
La clase Timer, perteneciente a Java Swing nos permite lanzar eventos regulares en intervalos de tiempo. Esta clase suele usarse para dibujar interfaces gráficas que requieran el redibujado sin interacción del usuario.

Ejercicios

Referencias y ampliación

UT12 - Gestión de bases de datos relacionales y persistencia

Bases de datos 1: Estableciendo el entorno de trabajo

Contenido

En esta unidad vamos a gestionar el acceso a bases de datos desde Java. El objetivo de la unidad es entender el API JDBC (Java Database Connectivity) y cómo realizar las operaciones básicas para trabajar con bases de datos.

El conocimiento adquirido en el módulo “Bases de datos” es importante para poder afrontar con éxito esta unidad de trabajo, pero nos centraremos en la parte que implica al lenguaje Java para realizar conexiones con bases de datos y realizar consultas elementales.

El API JDBC permite un acceso universal a los datos para el lenguaje Java. JDBC se conectará con diferentes drivers que nos permitirán interactuar con diferentes tecnologías de sistemas gestores de bases de datos, para poder obtener, borrar, actualizar o escribir datos.

El paquete java.sql y javax.sql (este segundo con extensiones y funcionalidad adicional) contienen las clases e interfaces del API de JDBC, que emplearemos para conectar con bases de datos y realizar operaciones con ellas.

Estableciendo el entorno de trabajo

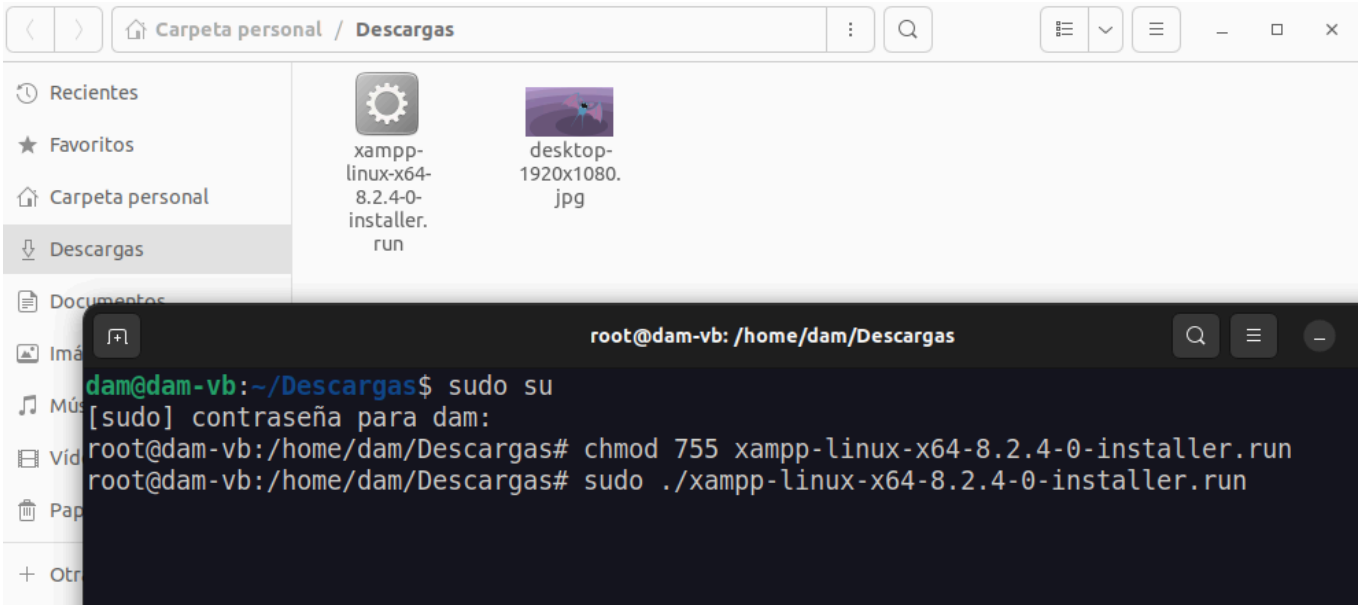
En esta unidad vamos a trabajar con el sistema gestor de bases de datos MySQL/MariaDB, y lo instalaremos empleando XAMPP, que es un paquete gratuito de Apache que tiene varias herramientas que nos permiten trabajar con estas bases de datos y herramientas para su gestión.

Pasos a seguir para la instalación:

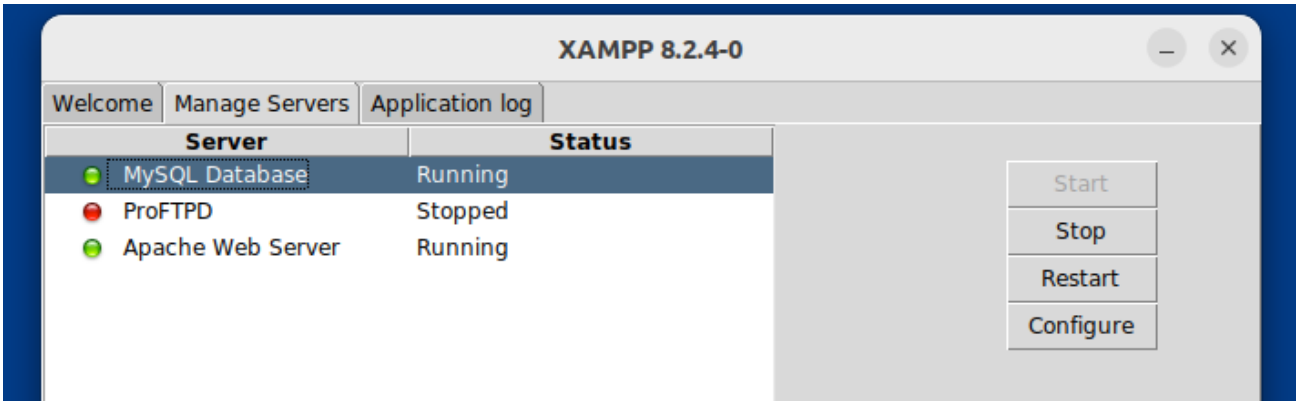
1. Descarga el instalador de XAMPP.
2. Accede a la carpeta de descarga y abre una terminal.
3. Con permisos de administración:
    a. chmod 755 [archivoXAMPP.run]

b. sudo ./[archivoXAMPP.run]

4. Acepta todo en la ventana que guía el proceso de instalación.
5. Cuando termine de instalar lanza la aplicación.



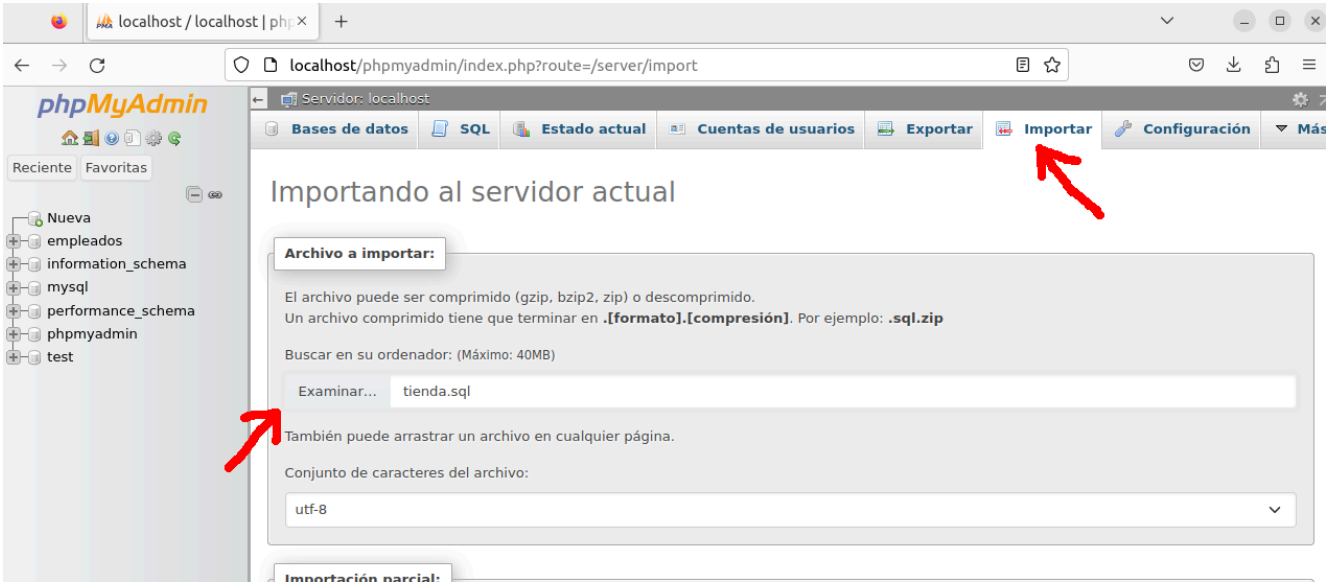
Para poder trabajar con las bases de datos debemos arrancar MySQL Database y Apache Web Server.



Con estos servicios arrancados podremos acceder al administrador con nuestro navegador, accediendo a: http://localhost/phpmyadmin/

Si se cierra el panel de control de XAMPP puedes abrirlo de nuevo tecleando en la terminal:
sudo /opt/lampp/manager-linux-x64.run

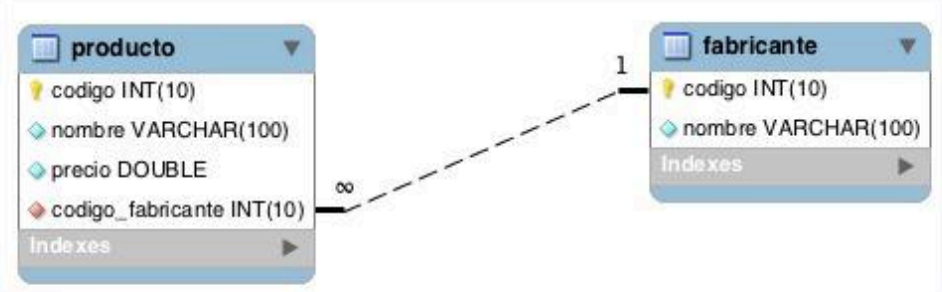
En el panel de control de PHPMyAdmin importamos las tres bases de datos (tienda, empleados y ventas) para trabajar durante la unidad:



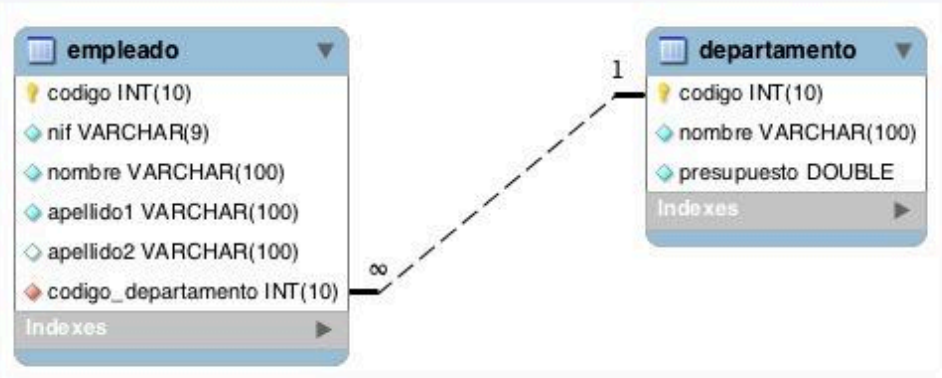
Pestaña importar -> seleccionar archivo -> botón importar (abajo)

Las bases de datos contienen datos de muestra y sus esquemas son los siguientes:

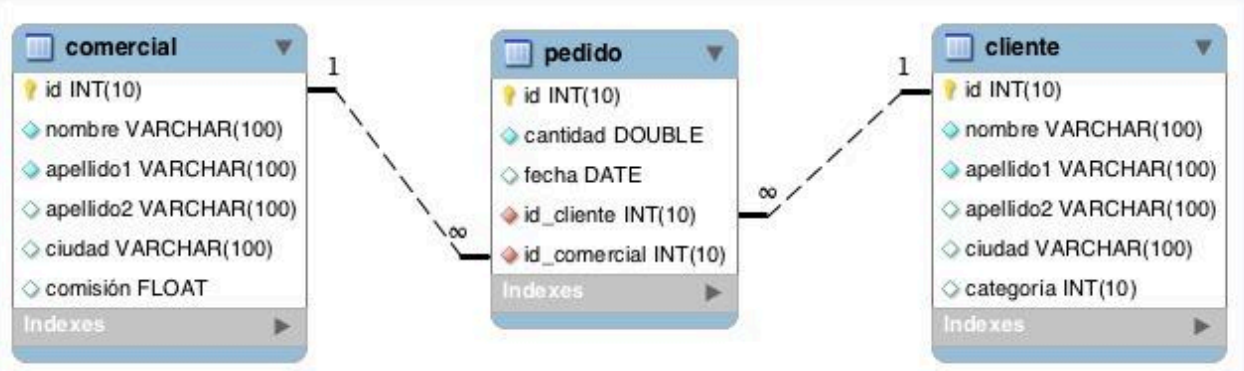
Tienda:



Empleados:

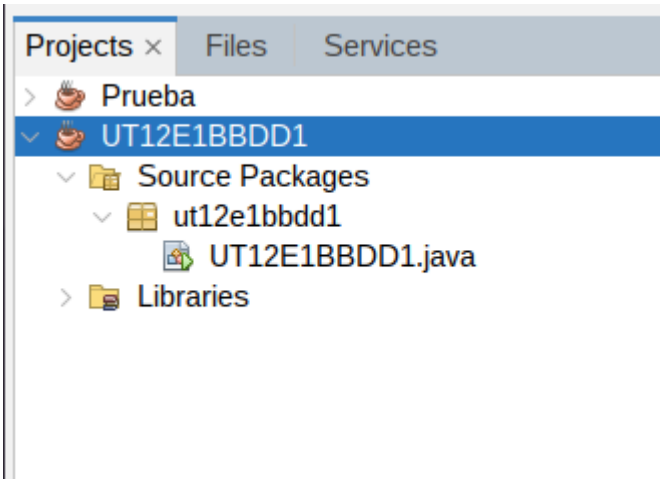


Ventas:

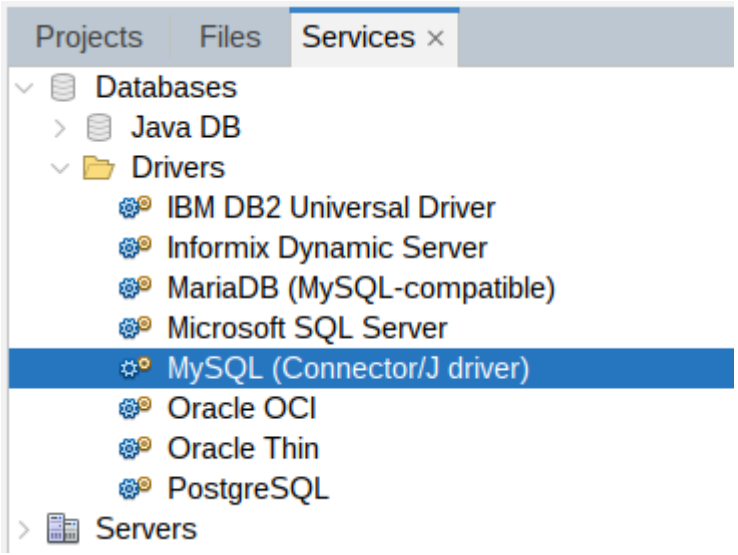


Para añadir un servicio a un proyecto en NetBeans (en este caso un driver de conexión con base de datos, debemos hacerlo en la pestaña Services).

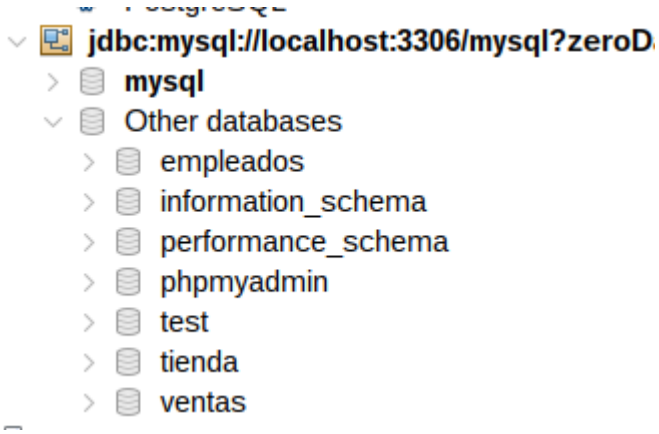
Lo descargamos de aquí:
https://dev.mysql.com/downloads/connector/j/
Descarga la versión “Platform independent”. Extrae la carpeta contenedora.



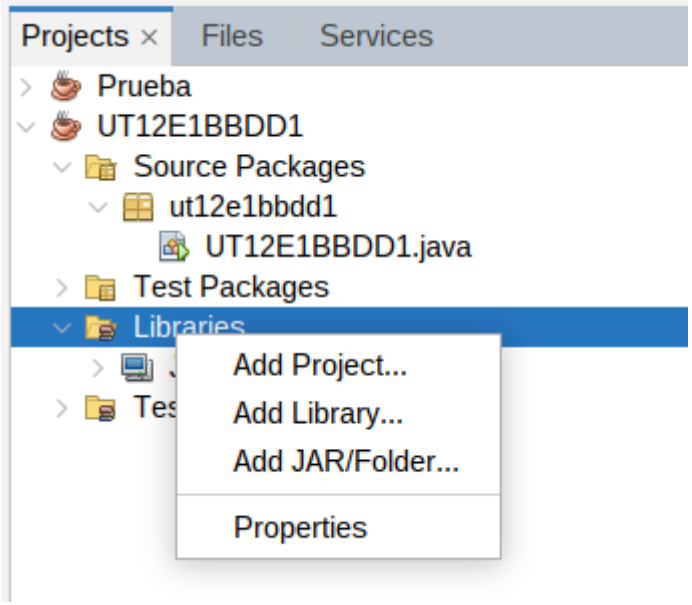
En la pestaña servicios debemos asociar el driver de conexión. Con el botón derecho selecciona configurar el driver y busca el archivo .jar dentro de la carpeta que se generó al bajar el driver y descomprimirlo.



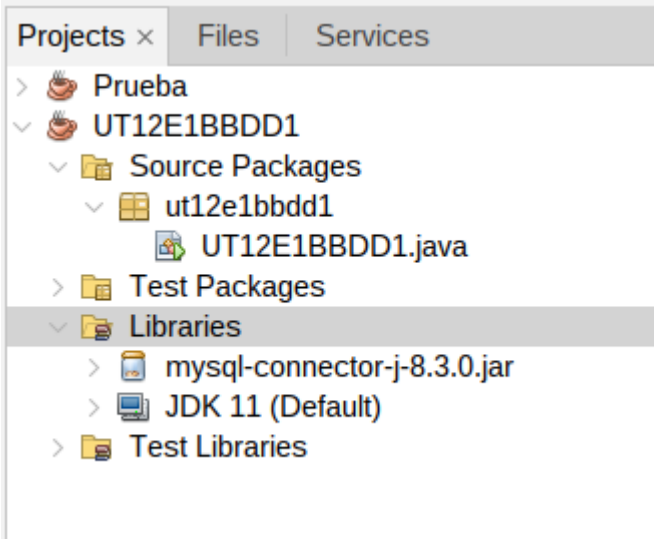
Si la conexión se realiza correctamente podrás ver todas las bases de datos de la siguiente forma:



Para añadir la librería al proyecto: botón derecho en Libraries -> Add JAR/Folder… y seleccionamos el jar del driver de conexión.



Obteniendo este resultado:



Si al ejecutar el código de esta unidad aparece por consola “Conexión establecida” es que está funcionando correctamente. De momento no es necesario entender el código.

Código de ejemplo

Ejercicios

Ejercicio 1.

Referencias y ampliación

Bases de datos 2: Clases e interfaces de JDBC

Contenido

El flujo de trabajo que debemos seguir para trabajar con bases de datos en Java con JDBC es el siguiente:

- Establecer una conexión (que usará un driver)
- Crear un Statement (sentencia).
- Ejecutar la consulta.
- Cerrar la conexión.

En esta lección vamos a entender el funcionamiento de las clases e interfaces principales de JDBC.

Clase DriverManager. Esta clase actúa como una interfaz entre las acciones realizadas por JDBC y los drivers (propietario del creador del sistema gestor de base de datos). Esta clase se encarga de registrar y desregistrar las conexiones del driver con la base de datos. Antes de comenzar a trabajar con la base de datos debemos registrar el driver.

Para el alcance de este curso, emplearemos el método getConnection, que recibe la dirección de la base de datos, el usuario y la contraseña con la que conectaremos. El método es estático, con lo cual podremos hacer uso de él sin necesidad de crear instancias de la clase. Este método puede generar una excepción de tipo SQLException, que deberemos gestionar.

Interfaz Connection. Esta interfaz representa una sesión de conexión con una base de datos. Esta interfaz es una factoría de sentencias (Statement y PreparedStatement), así como otros métodos de gestión de la base de datos.

Interfaz Statement. Esta interfaz provee de métodos para ejecutar consultas con la base de datos. Esta interfaz es una factoría de ResultStatement.

Interfaz PreparedStatement. Esta interfaz es una subinterfaz de Statement, y se emplea para ejecutar consultas parametrizadas. Las consultas parametrizadas tienen valores ? que se reemplazarán por valores al ser ejecutadas. PreparedStatement mejora la eficiencia ya que la consulta solo se ejecuta una vez, además es más segura de cara a inyecciones SQL.

Algunos métodos importantes de PreparedStetament son:

- setInt: para establecer un entero en un parámetro presentado por su índice.
- setString: para establecer una cadena de texto en un parámetro presentado por su índice.
- setFloat: para establecer un float en un parámetro presentado por su índice.
- setDouble: para establecer un double en un parámetro presentado por su índice.
- executeUpdate: ejecuta la consulta. Se emplea para las que no retornan valores por parte de la base de datos: create, drop, insert, update, delete, etc.
- executeQuery: ejecuta la consulta. Para aquellas que retornan datos (ResultSet).

Interfaz ResultSet. Esta interfaz se encarga de gestionar un conjunto de datos generados por una consulta. Tiene métodos para poder iterar por los resultados obtenidos y acceder a los datos. Algunos de los métodos más empleados son:

- next: para iterar sobre la siguiente fila.
- previous: para volver sobre la fila anterior.
- first: para ir al primer resultado.
- last: para ir al último resultado.
- absolute: para ir a un resultado, indicándose de forma numérica (por ejemplo acceder al quinto resultado).
- relative: para mover el cursor un número (positivo o negativo) de posiciones, desde la posición actual.
- getInt: para obtener datos de una columna como entero (a partir del número de columna o del nombre de la columna).
- String: para obtener datos de una columna como cadena (a partir del número de columna o del nombre de la columna).

SQLException. Algunos de los métodos de las clases e interfaces expuestas en esta lección pueden generar este tipo de excepción.

Código de ejemplo

Ejercicios

Ejercicio 1.

Referencias y ampliación

Bases de datos 3: Operaciones con bases de datos

Contenido

En esta lección vamos a trabajar con las clases e interfaces estudiadas en la lección anterior para realizar algunas de las operaciones más habituales en el trabajo con bases de datos.

Los ejemplos de esta lección se realizan sobre la base de datos "tienda" que importamos en la primera lección.

Estructura de la tabla "productos" de la base de datos "tienda":

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
1	**codigo**	int(10)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT
2	**nombre**	varchar(100)	utf8mb4_general_ci		No	*Ninguna*		
3	**precio**	double			No	*Ninguna*		
4	**codigo_fabricante**	int(10)		UNSIGNED	No	*Ninguna*		

Estructura de la tabla "fabricante" de la base de datos "tienda":

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
1	**codigo**	int(10)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT
2	**nombre**	varchar(100)	utf8mb4_general_ci		No	*Ninguna*		

Obteniendo datos

Para obtener datos lo haremos mediante una consulta de tipo SELECT. Para hacer esta consulta la debemos conformar, ejecutar y procesar los resultados obtenidos.

- Para conformar la consulta SQL la escribiremos y la guardaremos en una variable de tipo String.
- Para ejecutar la consulta crearemos una instancia de PreparedStatement pasándole el String creado y llamaremos a su método executeQuery.
- El método executeQuery retornará la respuesta de la base de datos y lo guardaremos en una instancia de ResultSet. Para procesar cada fila devuelta usaremos el método next y los métodos get para obtener los valores de las diferentes columnas (por índice o por nombre de columna).

Insertando datos

Para insertar datos lo haremos mediante una consulta de tipo INSERT INTO. Para hacer esta consulta la debemos conformar, ejecutar y comprobar si la inserción se ha realizado correctamente.

- Para conformar la consulta SQL la escribiremos y la guardaremos en una variable de tipo String. Los valores que pasemos a la consulta serán ?.
- Para establecer los valores reales usamos los métodos set de PreparedStatement con el tipo de dato asociado y la posición del ?. Ten en cuenta que la posición de los interrogantes es de izquierda a derecha y comienza en el índice 1.
- Para ejecutar la instrucción usamos el método executeUpdate, que retorna un entero, si este entero es 1 la inserción se ha realizado correctamente.

Actualizando datos

Para actualizar datos lo haremos mediante una consulta de tipo UPDATE. La operativa es muy similar a la de inserción.

Borrando datos
Para borrar datos lo haremos mediante una consulta de tipo DELETE. La operativa es muy similar a la de inserción.

Código de ejemplo

Ejercicios

Ejercicio 1.
Con la base de datos “tienda”. Obtén una lista de todos los productos con un precio inferior a 100 y preséntalo de esta forma: Nombre producto / nombre fabricante / precio
Después aplica un 20% de descuento a todos los productos de la tienda y vuelve a obtener la lista.
Finalmente devuelve el precio de todos los productos a su valor original.

Ejercicio 2.
Crea un nuevo fabricante en la base de datos “tienda”.
Lee el nombre de todos los empleados de la base de datos “empleados”. Crea un producto en la base de datos “tienda” por cada empleado, el nombre del empleado debe ser el nombre del producto y los tres últimos números del NIF debe ser el precio, el fabricante para todos los empleados será el que has creado al inicio del ejercicio.
Muestra la totalidad de la tabla de productos, conteniendo los productos y empleados nuevos, posteriormente borra todos los empleados y el fabricante creado e imprime de nuevo la tabla de productos.

Referencias y ampliación