Puntos clave Clase 2:

- Nuevo proyecto en IntelliJ v New Project

- Estructura de proyecto.

Desde SRC creo paquete.

Desde el paquete, botón secundario New Class (Menú contextual)

- Creación de nueva Clase : Archivo 1

- Creación de método “main “ Public Clasic void main…..

- Imprimir un texto “SOUT”

.

Los nombres de Archivos de Clase java uso mayúscula al inicio de cada palabra.

ArchivoEjemplo, NombreEjemplo, CocheProgram, etc

Las líneas terminan con punto y coma, salvo las que llevan llave.

Errores, se marcan con rojo en IntelIJ

Tipos de Comentarios.

// En la línea.

/\* De aquí hasta

\*/

*Comentarios JAVADOC:*

*Son orientados a generar documentación técnica.*

*Llevan dos asteriscos al principio.*

*/\*\* Por ejemplo para explicar que hace un bloque de código. Se suele poner a la altura de clase o método en la tabulación.*

package com.Bloquedos;  
*/\*\*  
 \* Clase de prueba para ceear un programa base  
 \*Estos mcomentarios se usan para clase  
 \*/*public class EjemplosComentarios {  
  
 */\*\* Comentarios de Métodos  
 \* punto de entrada a la aplicación java  
 \* permite ejecutar la aplicación  
 \** ***@param*** *args - Argumentos de entrada para la aplicación  
 \*/* public static void main(String[] args) {  
  
 // Esto es un comentario  
 //Estos comentarios se usan generalmente para uno mismo  
  
 System.*out*.println("Hola mundo");  
 /\*e  
 Esto es un comentario Multilinea  
 1- Crear la función suma.  
 2-Crear variables de tipo numérico.  
 3- Utilizar la fucnción para calcular la suma de los números  
 \*/  
 }  
}

Tipos de Variables:

Escribiremos los nombres de variables en formato **lowerCamelCase.** La primera letra se escribe en minúscula y, a continuación, si se utiliza más de una palabra, cada una de ellas empezaría con mayúscula.

package com.Bloquedos;  
/\*  
Recordar:  
Las Clases comienzan con Mayúsculas.  
Las variables y metodos con minuscula.  
Ojo con:  
true y false en los datos tipo boolean se escriben con minuscula en Java.  
 \*/  
public class TiposDeDatos {  
 public static void main(String[] args) {  
 //Tipo de dato nombre de dato = x;  
  
 //1. Numericos  
 //1.1 Enteros  
  
  
 byte variable1 = 5;  
 short variable2 = 10;  
 int variable3 = 30;  
 long variable4 = 100;  
 long numeroTelefono = 235505766;  
  
 //1.2 Decimales  
 float variable5 = 5.5f;  
 double variable6 =10.5d;  
 double precioSilla = 49.99;

//2.Boleano  
 boolean variable7 = false;  
 boolean variable8 = true; //Contro D permite duplicar línea.  
  
 // Texto  
 char variable9 ='a';  
 // Las tipos de variables anteriores se escriben con minúscula, son tipos PRIMITIVOS.  
 // A los tipos primitivos no se les puede asignar el valor null. Da error.  
 //String se escribe con mayúsculas porque es una Clase en Java.  
  
 String variable10 = "Hola Mundo";  
 }  
}

Una variable se define como:

Tipo Identificador = valor

Int A = 50

El envoltorio nos permite que una variable no tenga valor” null”. Por ejemplo para un campo no definido por el usuario en una base de datos.

Palabras Reservadas : Son las que aparecen con color.

Se puede buscar en un navegador java keywords y encontrar un listado.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Abstract | Continue | for | new | switch |
| assert\*\*\* | Default | goto\* | package | synchronized |
| Boolean | Do | if | private | this |
| Break | double | implements | protected | throw |
| Byte | else | import | public | throws |
| Case | enum\*\*\*\* | instanceof | return | transient |
| Catch | extends | int | short | try |
| Char | final | interface | static | void |
| Class | finally | long | strictfp\*\* | volatile |
| const\* | float | native | super | while |

|  |  |  |
| --- | --- | --- |
| \* |  | not used |
| \*\* |  | added in 1.2 |
| \*\*\* |  | added in 1.4 |
| \*\*\*\* |  | added in 5.0 |

*Fin Parte 1*

Operadores:

Aritméticos :+ - \* / %(resto )

package com.companySergin;  
  
public class Operadores {  
 public static void main(String[] args) {  
 //Operadores  
 //Aritméticos + - \* / para el resto %  
 int numero1 = 5;  
 int numero2 = 2;  
 int resultado = numero1 + numero2;  
 int resultado2 = numero1 - numero2;  
 int resultdo3 = numero1 \* numero2;  
 double resultado4 = numero1 / numero2;  
  
 //lógicos, relación, comparación, booleanos.  
  
 /\*  
 > mayor  
 >= mayor igual  
 > menor que  
 >= menor igual  
 == para comparar si son iguales  
 =! distinto  
 && and "y"  
 || or  
 ! negación. Ejemplo  
 !true quiere decit falso  
 \*/  
  
  
 //Operadores de Asignación  
 /\*  
 =  
 +=  
 -=  
 /=  
 \*=  
 %=  
  
 \*/  
  
 //Operadores de incremento:  
  
 // ++ se usa en bucle for  
 //decremento  
 // --  
 // concatenación  
 // + es para juntar dos cadenas de texto

package com.Bloquedos;  
  
public class operadores {  
 public static void main(String[] args) {  
 //Operadores  
 //Aritméticos + - \* / para el resto %  
 int numero1 = 10;  
 int numero2 = 20;  
 int resultadoSuma = numero1 + numero2;  
  
 // Se puede sumar directamente y devolver por pantalla dos níumeros.  
 System.*out*.println(numero1 + numero2);  
 System.*out*.println(resultadoSuma + 5.77);  
  
 // si es una operacion compleja se recomienda desdoblar y no repetir código  
 //Para renombrar variables utilizar clic derecho sobre la variable y refactor- rename  
  
 int resultadoResta = numero1 - numero2;  
 int resultado2 = numero1 - numero2;  
 int resultdo3 = numero1 \* numero2;  
 double resultado4 = numero1 / numero2;  
  
 //lógicos, relación, comparación, booleanos.  
  
 // Comparación  
 boolean resultado1 = numero1 > numero2;//false  
 System.*out*.println(resultado1);  
 boolean resultado2b = numero1 < numero2;// daría true  
 System.*out*.println(resultado2b);  
  
 /\* Operadores de Comparacion   
 > mayor  
 >= mayor igual  
 > menor que  
 >= menor igual  
 == para comparar si son iguales  
 =! distinto  
  
operadores Lógicos.  
 && and "y"  
 || or  
 ! negación. Ejemplo  
 !true quiere decit falso  
 \*/  
  
 boolean resultado3 =numero1 >5 && numero1 < 30;  
   
 int edad=17;  
   
 boolean carnetJoven = edad>=15 && edad <27;

Funciones.

*Una función es un bloque de código que se llama mediante su identificador.*

*Void es una palabra reservada que se utiliza para indicar que la función no devuelve ningún valor. Si no en vez de void va el yipo de dato (int, float, etc).*

*Static, indica que pertenece a la clase que estamos utilizando. Por ahora entender sólo eso.*

Las funciones las llamamos métodos cuando van asociadas a objetos.

package com.sergio;

public class Funciones {

public static void main(String[] args) {

holaMmundo(); //Llamada a funcion

holaMmundo("Sergio"); //Llamada a función con string

holaMmundo ("Sergio ", 47);

}

public static void holaMmundo(){ //Static indica que pertenece a l clase.

System.out.println("Hola Mundo");

}

protected static void holaMmundo(String name ){

System.out.println("Hola " + name);

}

protected static void holaMmundo(String name, int edad ){

System.out.println("Hola " + name);

}

//Funciones que reciben valor

public static int suma (int num1 , int num2){

return num1 + num2;

}

}

// Protected indica que la función solo puede ejecutarse desde el mismo paquete y clases hijas"

//La sobrecarga consiste en crear diferntes funciones con la misma signatura (Nombre), pero con diferentes parámetros.

//Por ejemplo la funcion holamundo está definida 2 veces, una sin parámetros y otra con Strings.}

// Lo que no se podría es llamar dos funciones con el mismo nombre si reciben el mismo parámetro.

//En éste último caso las estaría duplicando.

Archivo desde el que llamo otra función:

package com.sergio;

public class PruebaFunciones {

public static void main(String[] args) {

Funciones.holaMmundo(); //Llamo una función pública de otro archivo. (Nombre clase.funcion)

Funciones.holaMmundo("Sergio");

System.out.println(Funciones.suma(3,3));

}

}

Funciones Módulo Desarrollado:

package com.Bloquedos;  
  
public class funciones {  
 public static void main(String[] args) {  
 // opción 1: Función sin parámetro y sin tipo de retorno  
  
 *showMenu*(); //Llamo a la función  
  
 // opción 2: Función sin parámetro y con tipo de retorno  
 String menu = *getMenu*(); //La variable menu es igual a lo que devuelve la funcion.  
 System.*out*.println(menu); //Imprimimos el valor de la variable menu.  
  
 // opción 3: Función con parámetro y sin tipo de retorno.  
  
 *imprimirSaludo*("Open Boot Camp");  
  
 //Opción 4: Función con parámetro y con tipo de retorno.

String nombre ="Sergio";  
 String apellido ="Garaycochea";  
  
 String saludo = *obtenerSaludo*(nombre , apellido); //  
 System.*out*.println(saludo);  
  
 }  
  
  
 static void showMenu(){  
 System.*out*.println("Bienvenidos al E- comerce de zapatillas");  
 System.*out*.println("1. Ver Zapatillas");  
 System.*out*.println("2. Comprar Zapatillas");  
 System.*out*.println("3. Salir");  
 }  
  
 static String getMenu(){  
 //para devolver un valor uso Return  
 return "Bienvenidos al E- comerce de zapatillas \n 1. Ver Zapatillas.....";  
 }  
  
 private static void imprimirSaludo(String datos) {  
  
 System.*out*.println( "Buenos días " + datos);  
 }  
 static String obtenerSaludo(String nombre , String apellido ){  
 return "Buenas Tardes " + nombre + " " + apellido;  
  
 }  
}

package com.Bloquedos;  
  
public class Funciones2 {  
 public static void main(String[] args) {  
  
  
 int resultado1 = *Suma*(50,100);  
 int resultado2 = *Suma*(5,10);  
 int resultado3 = *Suma*(150,2100);  
 int resultado4 = *Suma*(550,1300);  
private static int Suma(int numero1, int numero2) {  
 return numero1 + numero2;  
 }  
}

SOBRECARGA DE FUNCIONES.

package com.Bloquedos;  
/\*  
Sobrecarga permite duplicar un método siempre y cuando tengan diferente numero /tipo parametro.  
 \*/  
public class Funciones3Sobrecarga {  
 public static void main(String[] args) {  
   
  
 }  
 private static int Suma(int numero1, int numero2) {  
 return numero1 + numero2;  
 }  
 private static int Suma(int numero1, int numero2, int numero3) {  
 return numero1 + numero2 + numero3;  
 }  
 private static double Suma(double numero1, int numero2) {  
 return numero1 + numero2;  
 }  
}

EJERCICIO

public class EjercicioBloqueFunciones {  
 public static void main(String[] args) {  
  
 double precio = 500;  
 double neto = *precioIva*(precio);  
 System.*out*.println(neto);  
 }  
  
 public static double precioIva(double precio) {  
 return precio \* 1.21;  
 }  
  
}

RECOMIENDA PAGINA codewars.com

ESTRUCTURAS DE CONTROL

IF

En el siguiente bloque solo se muestra el funcionamiento de la sentencia IF. Se muestra en la primera condición dos formas de expresarlo, una de éllas expresando lacoondición en un boolean.

public class SentenciaIf {  
 public static void main(String[] args) {  
 *int edad =19;*  
 if (edad >= 18) {  
 System.*out*.println("Es mayor de edad");  
 }  
 //Otra forma de realizar la acción anterior sería:  
  
 boolean esMayor= edad >= 18;//false  
 if(esMayor) {  
 System.*out*.println("Es mayor de edad");  
 }  
 if(edad <= 18) {  
 System.*out*.println("Es menor de edad");  
 }  
  
  
 }  
  
}

**IF ELSE**

//if else

// Ejemplo con booleano

public class IF\_Else {

public static void main(String[] args) {

// if (condicion) {

// codigo }

// Si el código es una sola línea se puden omitir las llaves.

boolean check = 5 < 10; // Devuelve true

if (check) {

System.out.println("Verdadero");

}

//Ejemplo con enteros.

int number1 = 5 ;

int number2 = 10;

int number3 = 20;

int number4 = 1000;

if (number1 < number2 && number2 < number3) {

System.out.println("verdadero");

}

else if (number3 < number4)

System.out.println("else if"); // Acá no utilicé llaves.

else{

}

System.out.println("fin");

}

}

IF ELSE

package estructurasControl.Condicionales;  
  
public class IfElse {  
 public static void main(String[] args) {  
 int edad = 19;  
 if (edad >= 18) {  
 System.*out*.println("Es mayor de edad"); //Si true entra aquí  
 } else {  
 System.*out*.println("Es menor de edad"); //Si false entra aquí  
  
 }  
 }  
}

IF ELSE

package estructurasControl.Condicionales;  
  
public class IfElse {  
 public static void main(String[] args) {  
 int edad = 19;  
 if (edad >= 18) {  
 System.*out*.println("Es mayor de edad"); //Si true entra aquí  
 } else {  
 System.*out*.println("Es menor de edad"); //Si false entra aquí  
 }  
 }  
}

IF ELSE IF

package estructurasControl.Condicionales;  
  
import com.sun.jdi.connect.Connector;  
  
public class ifElseIf {  
 public static void main(String[] args) {  
 String dia = "Lunes";  
  
 //Ejemplos comparar  
  
 boolean resultado = dia.equals("Martes");//la función equals se usa para comparar Strings.  
 //Para comparar números uso "==", pero para Strings variable.equals("valor")  
 boolean ResultadoCompararNumero = 5 == 5; //true  
  
 //If else if  
 if (dia.equals("Lunes")){  
 System.*out*.println("Es lunes :)");  
  
 } else if (dia.equals("Martes")) {  
 System.*out*.println("Es martes :)");  
  
 } else if (dia.equals("Miercoles")) {  
 System.*out*.println("Es Miercoles :)" );  
  
 } else if (dia.equals("Jueves")) {  
 System.*out*.println("Es Jueves :)");  
  
 }else System.*out*.println(" Es viernes o finde");  
  
  
  
 }  
}

SWICH CASE

Swich(Variable)

Case (condición 1) {código}

public class SwichCase {  
 public static void main(String[] args) {  
 String weater = "sunny";  
   
 switch (weater){  
 case "sunny":  
 System.*out*.println("El tiempo es soleado");  
 break;  
 case "Cloudy":  
 System.*out*.println("Nublado");  
 break;  
   
 default:  
 System.*out*.println("No se a podido identificar el clima");

**package estructurasControl.Condicionales;  
  
public class SwitchBreak {  
 public static void main(String[] args) {  
 String dia = "lunes";  
 switch (dia) {  
 case "lunes":  
 System.*out*.println("Lunes");  
 break;  
 case "martes":  
 System.*out*.println("Martes");  
 break;  
 case "miercoles":  
 System.*out*.println("Miercoles");  
 break;  
 case "jueves":  
 System.*out*.println("Jueves");  
 break;  
 case "viernes":  
 System.*out*.println("Viernes");  
 break;  
 case "sabado":  
 System.*out*.println("Sabado");  
 break;  
 case "domingo":  
 System.*out*.println("Domingo");  
 break;  
 default:  
 System.*out*.println("Dia no encontrado");   
 }  
 }}**

**BUCLES For**

package com.sergio;

public class ForLoop {

public static void main(String[] args) {

for(int i=0; i<=10; i++){

System.out.println(i +"Hola mundo");

}

}

}

BUCLE FOR Y USO CON ARRAYS

package estructurasControl.deRepeticion;  
  
public class bucleFor {  
 public static void main(String[] args) {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println("El valor de i es: " + i);  
 }  
  
 String[] nombres = {"Sergio, Clara, Mariana"}; //length 3  
 for (int i = 0; i < nombres.length; i++) {  
 System.*out*.println(nombres[i]);  
 }  
  
 int suma = 0;  
 int[] numeros = {5, 7, 8}; //Array de enteros, largo 3.}  
 for (int i = 0; i < numeros.length; i++) {  
 suma += numeros[i]; //sería lo mismo poner suma = suma + numeros[i]  
 }   
 System.*out*.println("La suma de los numeros es: " + suma);  
   
 }  
}

FOR EACH

package estructurasControl.deRepeticion;

public class bucleForEach {

public static void main(String[] args) {

String[] nombres = {"Sergio, Clara, Mariana"}; //length 3

for (String nombre: nombres) { // for(creovariable:Variable a usar)

System.out.println(nombres);

}

int [] numeros = {5,10,15};

for (int numero: numeros) {

System.out.println(numero);

}

}

}

Bulces While //while (Condición) {Código}

package com.sergio;  
  
public class bucle\_While {  
 public static void main(String[] args) {  
  
 int count = 0; //Contador  
 while(count < 10){  
 System.*out*.println("Hola mundo " +count);  
 count ++;  
 }  
 System.*out*.println("fin");  
 }  
 }

//lAS VARIABLES CREADAS DENTRO DEL BUCLE EXISTEN SOLO DENTRO DE ÉL  
  
package estructurasControl.deRepeticion;  
  
public class BucleWhile {  
 public static void main(String[] args) {  
  
 //BUCLE INFINITO.  
 // while (true){  
 // System.out.println("Hola");// Bucle infinito.  
 // }  
  
 //BUCLE CORTADO  
 int contador = 0;  
 while (contador < 10) {  
  
 contador++;  
 if (contador == 5) {  
 break; //podemos romper el código antes que se de la condición.  
  
 }  
 }  
 System.*out*.println("valor del contador " + contador);  
  
 //CONTINUE  
  
 contador = 0;  
 while (contador < 10){  
 contador ++ ;  
 if (contador == 5) {  
 continue; // Continue indica que pase a la siguiente iteracción  
 } // El valor 5 NO SE IMPRIME !  
 System.*out*.println("valor del contador "+contador);   
 }  
 }  
}

BREAK Y CONTINUE

public class break\_Continue {  
 public static void main(String[] args) {  
 int count = 0; //Contador  
 while(count < 10) {  
 count ++;  
 if (count == 6)  
 break; //rompe el código y sale  
 // continue; //si la varible vale 6 pasa a la siguiente iteraccion  
 System.*out*.println("Hola mundo " +count);  
  
 }  
  
 System.*out*.println("Hola mundo " +count);  
  
 System.*out*.println("fin");  
 }  
}

}

Creación de Clases. Clase Coche.

package com.sergio;  
public class Coche {  
  
  
//1-Atributos :  
 String color;  
 String fabricante;  
 String modelo;  
 Double peso;  
 Double largo;  
 Integer velocidad = 0;  
  
   
  
//2-Constructores Son los métodos que nos permiten crear objetos de ésta clase.  
 //Si no añadimos ningún constructor, habrá por defecto un constructor vacío que nos permitirá crear objetos   
 //sin asignar valores a los parámetros.  
public Coche(String color,String fabricante, String modelo, Double peso, Double largo) {  
 this.color = color;  
 this.fabricante=fabricante;  
 this.modelo=modelo;  
 this.peso=peso;  
 this.largo=largo;  
  
}  
El constructor es una función. Tiene la particularidad de escribir su primera letra en mayúscula.

Recibe valores desde fuera. Para asignarlos usamos la palabra reservada “this” con la que nos referimos al atributo   
El dato que ingresa a través del String fabicante se va a “asignar” al atributo fabricante luego de la palabra this.

Public Coche (String color, String fabricante,………..

this.color = color;

This.fabricante = fabricante;  
//3-Comportamientos  
  
public void acelerar(Integer cantidad) {  
 this.velocidad += cantidad;  
}  
  
 @Override //boton secundario para generar  
   
 public String toString() {  
 return "Coche{" +  
 "color='" + color + '\'' +  
 ", fabricante='" + fabricante + '\'' +  
 ", modelo='" + modelo + '\'' +  
 ", peso=" + peso +  
 ", largo=" + largo +  
 ", velocidad=" + velocidad +  
 '}';  
 }  
}

CREAR OBJETOS.

package com.sergio;  
  
public class CocheMain {  
 public static void main(String[] args) {  
 String coche = "alfa Romeo";  
  
 Coche cocheobj = new Coche();  
  
 //Control +P me muestra los parámetros que debo definir.  
 Coche cocheobj2 = new Coche("rojo", "Honda", "Cicic", 1430.45, 5.4);  
  
 cocheobj2.acelerar(50);  
  
 System.*out*.println(cocheobj2);  
  
 cocheobj2.peso = 1350.8;  
  
 System.*out*.println(cocheobj2);  
  
 //Debajo creo un objeto Coche electrico.  
 CocheElectrico cocheElectrico = new CocheElectrico();  
 cocheElectrico.motorElectico = "Ejemplo Motor";  
 cocheElectrico.color = "verde";  
 cocheElectrico.fabricante = "Honda";  
 cocheElectrico.modelo ="civic";  
 System.*out*.println(cocheElectrico);  
  
 }  
}

PONEMSOEMOS MAIN PARA INDICAR QUE ES EL PUNTO DE ENTRADA A LA APLICACIÓN

HERENCIA:

package com.sergio;  
  
public class CocheElectrico extends Coche {  
  
 String motorElectico;  
 // Aptovechando el concepto de sobrecarga de funciones vamos a crear diferentes tipos de coches electricos.  
  
 public CocheElectrico() { //Constructor Vacío.  
  
 }  
  
 public CocheElectrico(String motorElectico) {  
 this.motorElectico = motorElectico;  
 }  
  
 @Override // permite imprimir el valor de las varibles  
 public String toString() {  
 return "CocheElectrico{" +  
 "motorElectico='" + motorElectico + '\'' +  
 ", color='" + color + '\'' +  
 ", fabricante='" + fabricante + '\'' +  
 ", modelo='" + modelo + '\'' +  
 ", peso=" + peso +  
 ", largo=" + largo +  
 ", velocidad=" + velocidad +  
 '}';  
 }  
}

EL METODO SUPER

package com.sergio;  
  
public class CocheMain {  
 public static void main(String[] args) {  
 String coche = "alfa Romeo";  
  
 Coche cocheobj = new Coche();  
  
 //Control +P me muestra los parámetros que debo definir.  
 Coche cocheobj2 = new Coche("rojo", "Honda", "Cicic", 1430.45, 5.4);  
  
 cocheobj2.acelerar(50);  
  
 System.*out*.println(cocheobj2);  
  
 cocheobj2.peso = 1350.8;  
  
 System.*out*.println(cocheobj2);  
  
 //Debajo creo un objeto Coche electrico.  
 CocheElectrico cocheElectrico = new CocheElectrico();  
 cocheElectrico.motorElectico = "Ejemplo Motor";  
 cocheElectrico.color = "verde";  
 cocheElectrico.fabricante = "Honda";  
 cocheElectrico.modelo ="civic";  
 System.*out*.println(cocheElectrico);  
//Herencia. El método super: permite acceder al constructor de la clase base.  
  
 CocheElectrico cocheElectrico2 = new CocheElectrico("Azul" ,"Alfa", "romeo", 1500d, 4.99,  
 "Motor de 5 KW");  
 System.*out*.println(cocheElectrico2);  
 }  
}

SOBREESCRITURA.

SE MODIFICÓ PARTE DE LOS ARCHIVOS ANTERIORES

package com.sergio;  
  
public class CocheElectrico extends Coche {  
  
 String motorElectico;  
 // Aptovechando el concepto de sobrecarga de funciones vamos a crear diferentes tipos de coches electricos.  
  
 public CocheElectrico() { //Constructor Vacío.  
  
 }  
 public CocheElectrico(String motorElectico) {  
 this.motorElectico = motorElectico;  
 }  
  
 public CocheElectrico(String color, String fabricante, String modelo, Double peso, Double largo, String motorElectico){  
 // si yo pusiera this .color= color.  
 // this.......  
 //Estaría duplicando código.  
 //El método SUPER me permite reutilizar ese Código  
 //El metodo SUPER invoca un CONSTRUCTOR DE LA CLASE SUPERIOR.  
  
 super(color, fabricante, modelo, peso,largo);  
 this.motorElectico = motorElectico;  
  
 }  
 @Override //le indica al compilador que verifique si hay un metodo acelerar en la clase superior  
 public void acelerar (Integer cantidad) {  
 Integer cantidadAjustada = cantidad \* 2; //Aquí estamos modificando el valor que ingresa al método  
 super.acelerar (cantidadAjustada); //Con el método super accedemos al método acelerar de la clase superior.  
 }  
 @Override  
 public String toString() {  
 return "CocheElectrico{" +  
 "motorElectico='" + motorElectico + '\'' +  
 ", color='" + color + '\'' +  
 ", fabricante='" + fabricante + '\'' +  
 ", modelo='" + modelo + '\'' +  
 ", peso=" + peso +  
 ", largo=" + largo +  
 ", velocidad=" + velocidad +  
 '}';  
 }  
}

POLIMORFISMO:

Cuando no lo utilizamos estamos utilizando la clase concreta. La clase hija.

Cuando lo utilizamos estamos utilizando la clase superior.

Esto será útil cuando utilicemos listas de arrays en bucles donde todos los coches deban ser tratados por igual.

package com.sergio;  
  
public class PolimorfismoMain {  
 public static void main(String[] args) {  
  
 Coche coche1 = new Coche();  
 CocheElectrico coche2 = new CocheElectrico();  
 CocheHibrido coche3 = new CocheHibrido();  
  
 // Polimorfismo:  
  
 Coche coche4 = new CocheElectrico(); // Genera el objeto coche 4 como el tipo padre coche.  
 Coche coche5 = new CocheHibrido();  
 // Coche coche5 = new CocheHibrido(); no se por que da error  
 if (coche4 instanceof CocheElectrico) { // INSTANCEOF permite identificar el tipo de clase  
  
 System.*out*.println("Coche Electrico");  
 }  
 if (coche4 instanceof CocheHibrido) { // INSTANCEOF permite identificar el tipo de clase  
 System.*out*.println("Coche Hibrido");   
 }  
 }  
}

Clase abstracta y utilización.

Es un tipo de clase que no puede instanciarse y que utilizamos normalmente cuando tenemos herencia.

En los ejemplos la Clase coche es la clase padre de las que heredan coche eléctrico y coche híbrido.

En las clases abstractas (Las cuales pueden tener atributos, funciones) solo pueden instanciarse a través de las clases hijas. Es decir en nuestro ejemplo si declaramos a Coche como clase abstracta, nos daría error en todo código dónde dicha clase fue instanciada.

Para hacerlo usamos la palabra reservada abstract

public abstract class Coche{

Programas y teoría de Programación Orientada a Objetos. Curso Inicial.

CREACION DE OBJETOS. EJEMPLO1

package poo.clases;  
import poo.herencia.Camion;  
import poo.herencia.Coche;  
import poo.herencia.Motocicleta;  
  
public class Main {  
 public static void main(String[] args) {  
/\*  
1- CLASE Y OBJETOS  
Para crear un objeto:  
Clase identificador = new Clase();  
 Clase() es la llamada al constructor.  
 \*/  
 // Crear un objeto utilizando un constructor vacío.  
 Vehiculo toyotaPrius = new Vehiculo(); // me posiciono dentro de los parentesis y CTRL P me da las opciones.  
  
 // Crear un objeto utilizando el constructor con parámetros  
 Motor motorGTI = new Motor("GTI", 190, 459.0, 6);

Vehiculo fordMondeo = new Vehiculo("Ford", "Mondeo", 2.1, 2010, false, 0, motorGTI);  
 System.*out*.println(fordMondeo.fabricante);  
 System.*out*.println(fordMondeo.year);  
 System.*out*.println(fordMondeo.speed);  
 fordMondeo.acelerar(50);  
 System.*out*.println(fordMondeo.speed);  
  
 //2- HERENCIA  
  
 Motocicleta kawasakiNinja = new Motocicleta();  
 kawasakiNinja.fabricante = "Kawasaky";  
  
 System.*out*.println("Fin de programa");  
  
 //3-POLIMORFISMO  
 //Vehiculo es una forma de la que se desprenden otras con otras formas  
 Vehiculo vehiculo;  
 vehiculo = new Motocicleta();  
 vehiculo.acelerar(50);  
  
 vehiculo = new Coche();  
 vehiculo.acelerar(50);  
  
 vehiculo = new Camion();  
 vehiculo.acelerar(50);  
  
// 4 CLASES ABSTRACTAS NO PUEDEN INSTANCIARSE  
 /// mODIFICADOR sTATICT - QUEDE ACA 5.11  
 }  
}

package poo.clases;  
  
public class Motor {  
 String modelo;  
 int caballo;  
 double parNn;  
 int numCilindros;  
  
  
 // Constructores.  
 public Motor(){  
  
 }  
  
 public Motor( String modelo, int caballo, double parNn, int numCilindros) {  
 this.modelo = modelo;  
 this.caballo = caballo;  
 this.parNn = parNn;  
 this.numCilindros = numCilindros;  
 }  
 // Metodos  
}

package poo.clases;  
/\*  
Clase Base opadre o Superclase  
 \*/  
public class Vehiculo {  
 //1.Atributos  
  
 protected String fabricante;  
 protected String modelo;  
 protected Double cc; //doubles es un tipo primitivo. Double (con mayúscula) es un tipo de envoltorio que permite valores nulos.}  
 protected int year;  
 protected boolean sport;  
 protected int speed ;  
 Motor motor; //Defino un objeto dentro de otro para reducir la complegidad del código.  
  
 //2.Constructores: Son métodos especiales que nos permiten construir el objeto a través de ésta plantilla  
 //Por convención son publicos, el nombre que recibe el constructor es el mismo que el de la clasey luego  
 //Paréntesis ()y llaves.  
 public Vehiculo() {  
 //Constructor vacío. Si creo el objeto no tendrá valores asignados.  
 }  
 public Vehiculo(String fabricante, String modelo, Double cc, int year, boolean sport, int speed, Motor motor) {  
 this.fabricante = fabricante;  
 this.modelo = modelo;  
 this.cc = cc;  
 this.year = year;  
 this.sport = sport;  
 this.speed = speed;  
 this.motor = motor;  
 }  
 public Vehiculo(String fabricante, String modelo, Double cc, int year, boolean sport) {  
 this.fabricante = fabricante;  
 this.modelo = modelo;  
 this.cc = cc;  
 this.year = year;  
 this.sport = sport;  
 this.speed = 0;  
 }  
  
  
 // Debajo con sobrecarga se crea un constructor de sólo 2 atributos.  
 public Vehiculo(String fabricante, String modelo) {  
 this.fabricante = fabricante;  
 this.modelo = modelo;  
 }  
  
  
 //3. Métodos (comportamientos  
  
 public void acelerar(int quantity){  
 this.speed += quantity;  
 }  
  
  
 //getter y setter  
  
 // to string  
  
}

Ejemplo 2

**package poo.clases;**import poo.herencia.Camion;  
import poo.herencia.Coche;  
import poo.herencia.Motocicleta;  
  
public class Main {  
 public static void main(String[] args) {  
/\*  
1- CLASE Y OBJETOS  
Para crear un objeto:  
Clase identificador = new Clase();  
 Clase() es la llamada al constructor.  
 \*/  
 // Crear un objeto utilizando un constructor vacío.  
 Vehiculo toyotaPrius = new Vehiculo(); // me posiciono dentro de los parentesis y CTRL P me da las opciones.  
  
  
 // Crear un objeto utilizando el constructor con parámetros  
 Motor motorGTI = new Motor("GTI", 190, 459.0, 6);  
  
  
 Vehiculo fordMondeo = new Vehiculo("Ford", "Mondeo", 2.1, 2010, false, 0, motorGTI);  
 System.*out*.println(fordMondeo.fabricante);  
 System.*out*.println(fordMondeo.year);  
 System.*out*.println(fordMondeo.speed);  
 fordMondeo.acelerar(50);  
 System.*out*.println(fordMondeo.speed);  
  
 //2- HERENCIA  
  
 Motocicleta kawasakiNinja = new Motocicleta();  
 kawasakiNinja.fabricante = "Kawasaky";  
  
 System.*out*.println("Fin de programa");  
  
 //3-POLIMORFISMO  
 //Vehiculo es una forma de la que se desprenden otras con otras formas  
 Vehiculo vehiculo;  
 vehiculo = new Motocicleta();  
 vehiculo.acelerar(50);  
  
 vehiculo = new Coche();  
 vehiculo.acelerar(50);  
  
 vehiculo = new Camion();  
 vehiculo.acelerar(50);  
  
// 4 CLASES ABSTRACTAS NO PUEDEN INSTANCIARSE  
 /// mODIFICADOR sTATICT - QUEDE ACA 5.11  
 }  
}

package poo.clases;  
/\*  
Clase Base opadre o Superclase  
 \*/  
public class Vehiculo {  
 //1.Atributos  
  
 protected String fabricante;  
 protected String modelo;  
 protected Double cc; //doubles es un tipo primitivo. Double (con mayúscula) es un tipo de envoltorio que permite valores nulos.}  
  
 protected int year;  
 protected boolean sport;  
 protected int speed ;  
  
 Motor motor; //Defino un objeto dentro de otro para reducir la complegidad del código.  
  
 //2.Constructores: Son métodos especiales que nos permiten construir el objeto a través de ésta plantilla  
 //Por convención son publicos, el nombre que recibe el constructor es el mismo que el de la clasey luego  
 //Paréntesis ()y llaves.  
 public Vehiculo() {  
 //Constructor vacío. Si creo el objeto no tendrá valores asignados.  
 }  
  
 public Vehiculo(String fabricante, String modelo, Double cc, int year, boolean sport, int speed, Motor motor) {  
 this.fabricante = fabricante;  
 this.modelo = modelo;  
 this.cc = cc;  
 this.year = year;  
 this.sport = sport;  
 this.speed = speed;  
 this.motor = motor;  
 }  
  
 public Vehiculo(String fabricante, String modelo, Double cc, int year, boolean sport) {  
 this.fabricante = fabricante;  
 this.modelo = modelo;  
 this.cc = cc;  
 this.year = year;  
 this.sport = sport;  
 this.speed = 0;  
 }  
  
  
 // Debajo con sobrecarga se crea un constructor de sólo 2 atributos.  
 public Vehiculo(String fabricante, String modelo) {  
 this.fabricante = fabricante;  
 this.modelo = modelo;  
 }  
  
  
 //3. Métodos (comportamientos  
  
 public void acelerar(int quantity){  
 this.speed += quantity;  
 }  
  
  
 //getter y setter  
  
 // to string  
  
}

package poo.herencia;  
  
import poo.clases.Vehiculo;  
  
public class Camion extends Vehiculo {  
  
 double capacidadCarga;  
  
}

package poo.herencia;  
  
import poo.clases.Vehiculo;  
  
public class Coche extends Vehiculo {  
  
 int numPuertas;  
}

package poo.herencia;  
  
import poo.clases.Motor;  
import poo.clases.Vehiculo;  
/\*  
Clase derivada o clase hija o subclase .  
 \*/  
public class Motocicleta extends Vehiculo {  
 boolean baul;  
  
 public Motocicleta(){  
  
 }  
 //Constructor creado con boton secunsario, generate constructor.  
 public Motocicleta(String fabricante, String modelo, Double cc, int year, boolean sport, int speed, Motor motor, boolean baul) {  
 super(fabricante, modelo, cc, year, sport, speed, motor);  
 this.baul = baul;  
 }  
}

Interfaces:

Es un mecanismo de abstracción que nos permite desacoplar el código y nos permite hacerlo mas flexible y mantenerlo mucho mejor.

Creamos la interfaz y ésta nos permitirá crear métodos para trabajar con los objetos. Ejemplo coches.

public interface CocheService {  
 public Coche crearCocheDemo(); //ACochequí van paréntesis porque los métodos de interfaz no tienen cuerpo.  
} //Aquí tenemos el nombre del método. El mismo se implementará en otra parte.  
//En el cursso anterior se explicaba que es una forma de pasarle al programador que se va a hacer y el decide el como.

Ejemplo Implementaciones de interfaz

// Acá se imlementa el método declarado en la interfaz  
public class CocheServiceClassicImpl implements CocheService {  
  
  
 @Override  
 public Coche crearCocheDemo() {  
 System.*out*.println("Creando coche Clasico");  
 return new CocheElectrico();  
 }  
}

public class CocheServiceSportImpl implements CocheService{  
 @Override  
 public Coche crearCocheDemo() {  
 System.*out*.println("creando coche carrera");  
 return new CocheHibrido();  
 }  
}

MAIN Interfaces

Public class InterfacesMain {  
 public static void main(String[] args) {  
 CocheService service1 = new CocheServiceSportImpl();  
 CocheService service2 = new CocheServiceSportImpl();  
 Coche coche1 = service1.crearCocheDemo();  
 Coche coche2 = service2.crearCocheDemo();  
 }  
}

Interfaces (JAVA BASICO).

Las interfaces nos permiten proveer abstracción. Poder hacer que un código que dependa de otro no esté acoplado

Da un nuevo ejemplo de la gestión de una base de datos de empleados.

Main Interfaz Implementaciones

EmpleadoCRUDV1

Interfaz

Main

Si no tengo la Interfaz en el medio es muy complejo

EmpleadoCRUDV2

Actualizar el código. Con la interfaz desacoplo la versión uno

De Empleado CRUD y acoplo la 2

1. Ejemplo sin la Interfaz.

package poo.sinInterfaces;  
  
import java.util.List;  
  
public class Main {  
 public static void main(String[] args) {  
  
  
 EmpleadoCRUD empleadoCRUD = new EmpleadoCRUD();  
 Empleado juanito = new Empleado("Juanito", 30, 2000.0, true);  
 Empleado patricia = new Empleado("Patricia", 30, 2000.0, true);  
 Empleado roberto = new Empleado("Roberto", 30, 2000.0, true);  
 //GUARDAR EMPLEADOS  
  
 empleadoCRUD.save(juanito);  
 empleadoCRUD.save(patricia);  
 empleadoCRUD.save(roberto);  
  
 //CONSULTAR EMPLEADOS  
 List<Empleado> empleados = empleadoCRUD.findAll();  
 System.*out*.println(empleados);

Objeto Empleado

package poo.sinInterfaces;  
  
public class Empleado {  
  
 //1 - Atributos  
  
 String nombre;  
  
 int edad;  
 double salario;  
 boolean alta;  
  
 // 2 - Constructores ES IDONEO CREAR UNO VACIO Y UNO CON TODOS LOS PARAMETROS.  
 public Empleado() {  
  
 }  
  
 public Empleado(String nombre, int edad, double salario, boolean alta) {  
 this.nombre = nombre;  
 this.edad = edad;  
 this.salario = salario;  
 this.alta = alta;  
 }  
  
//Metodos Genero método toString para poder visualizar los empleados. Si no muestra la referncia en memoria.  
  
  
 @Override  
 public String toString() {  
 return "Empleado{" +  
 "nombre='" + nombre + '\'' +  
 ", edad=" + edad +  
 ", salario=" + salario +  
 ", alta=" + alta +  
 '}';  
 }  
}

Empleado CRUD. Emulacion base de datos

package poo.sinInterfaces;  
  
import java.util.ArrayList;  
import java.util.List;  
//Version 1  
public class EmpleadoCRUD {  
 /\*  
  
 Emulamos que guardamos el empleado en una base de datos en ésta estructura.  
 Usamos arraylist porque el array es estático, en cambio el que usaremos permite borrar, agregar, etc.  
  
 CRUD VIENE DE Operaciones CRUD: (Se utiliza muvho en desarrollo de software  
 C create  
 R read  
 U update  
 D delete  
  
\*/  
  
 //Estructura de datos  
 List <Empleado> empleados = new ArrayList<>();  
 //List es una interface que nos permite utilizar una estructura que se comporta como un array normal pero DINAMICO  
  
 //guardar un empleado. Create  
  
 //Operaciones: CRUD:  
 //Create  
 public void save (Empleado empleado) {  
 empleados.add(empleado);  
 System.*out*.println("Empleado guardado");  
 }  
 public List<Empleado> findAll(){  
 return empleados;

2 -Ejemplo con Interfaz

Main

public class Main {  
 static EmpleadoCRUD *empleadoCRUD* = new EmpleadoCRUDExcel();  
 //static EmpleadoCRUD empleadoCRUD = new EmpleadoCRUDMySQL(); Solo cambiando una linea podria alternar de escel a SQL  
  
 public static void main(String[] args) {  
  
 *empleadoCRUD*.findAll();  
 *empleadoCRUD*.save(new Empleado());  
  
 }  
}

INTERFACE

public interface EmpleadoCrud{  
  
//En la INTERFAZ SE declaran los métodos, no se implementan, por eso llevan ";" al final.  
 //La interfaz funciona como un contrato.Dice lo que hay que hacer pero no como.  
  
 void save(Empleado empleado);//Declaro método save que recibe una clase del tipo Empleado  
  
 List<Empleado> findAll(); //Declaro el método findAll que devuelve lista.  
  
 void delete (Empleado empleado); //La función delete recibe un objeto del tipo Emleado y no devuelve nada  
}

Implementacion para “Supuesta base de datos Excel

import java.util.List;  
  
  
//No vamos a completar el código. Eso es complejo y se verá mas adelante.  
//Por ejemplo en SAVE habría que abrir un archivo excel y guardar datos.  
public class EmpleadoCRUDExcel extends EmpleadoCRUD {  
 @Override  
 public void save(Empleado empleado) {  
  
 }  
  
 @Override  
 public List<Empleado> findAll() {  
 return null;  
 }  
  
 // @Override  
 //public void delete(Empleado empleado) {  
  
 //}  
}

EJEMPLO DE SERGIO COLECTIVOS

OBJETO Colectivo

public class Colectivo {  
  
 int numero;  
 String Marca;  
 int km;  
 int capacidad;  
 int year;  
  
 public Colectivo(int numero, String marca, int km, int capacidad, int year) {  
 this.numero = numero;  
 Marca = marca;  
 this.km = km;  
 this.capacidad = capacidad;  
 this.year = year;  
 }  
 @Override  
 public String toString() {  
 return "Colectivo{" +  
 "numero=" + numero +  
 ", Marca='" + Marca + '\'' +  
 ", km=" + km +  
 ", capacidad=" + capacidad +  
 ", year=" + year +  
 '}';  
 }  
}

INTERFACE:

import java.util.List;  
  
public interface ColectivoCRUD {  
  
 public void create (Colectivo colectivo);  
 List<Colectivo> read();  
  
 void update (Colectivo colectivo);  
 void delete (Colectivo colectivo);   
}

IMPLEMENTACION DE LA INTERFACE

import java.util.ArrayList;  
import java.util.Collection;  
import java.util.Collections;  
import java.util.List;  
  
public class ColectivoCRUDimp implements ColectivoCRUD {  
 List <Colectivo> colectivos = new ArrayList<>();  
 @Override  
 public void create(Colectivo colectivo) {  
 colectivos.add(colectivo);  
 System.*out*.println("Colectivo guardado");  
  
 }  
  
 @Override  
 public List<Colectivo> read() {  
 ;  
 return colectivos ;  
  
  
 }  
  
 @Override  
 public void update(Colectivo colectivo) {  
  
 }  
  
 @Override  
 public void delete(Colectivo colectivo) {  
  
 }  
}

MAIN Colectivos…..

public class Main {  
 public static void main(String[] args) {  
  
 ColectivoCRUD Colectivo = new ColectivoCRUDimp();  
  
 Colectivo colectivo1 = new Colectivo(22,"Mercedes",200000, 38, 2020);  
 Colectivo colectivo2 = new Colectivo(23,"Iveco",30000, 38, 2022);  
 Colectivo colectivo3 = new Colectivo(24,"Mercedes",1200000, 38, 2020);  
 Colectivo colectivo4 = new Colectivo(25,"Mercedes",130000, 38, 2020);  
  
 //Creación  
  
 Colectivo.create(colectivo1);  
 Colectivo.create(colectivo2);  
 Colectivo.create(colectivo3);  
 Colectivo.create(colectivo4);  
  
  
 // Read  
  
 System.*out*.println( Colectivo.read());  
  
 }  
}

Creacion de Clases (Visto anteriormente en Polimorfismo.

public class polimorfismo {  
 public static void main(String[] args) {  
 // Coche coche1 = new Coche();  
 CocheElectrico coche2 = new CocheElectrico();  
 CocheHibrido coche3 = new CocheHibrido();  
  
 // Polimorfismo:  
// Las clases anteriores se instancian a bajo nivel como tipo Coche, Coche Eléctico e Híbrido.  
  
  
 //Lo que se hace debajo es instanciatlas como una clase superior. Es decir Tipo Coche.  
  
  
 Coche coche4 = new CocheElectrico(); // Genera el objeto coche 4 como el tipo padre coche.  
 Coche coche5 = new CocheHibrido();

Técnicas de Refactoring:

El refactoring o refactorización de código consiste en adaptar un código para hacerlo mucho mas legible y que se pueda mantener mucho mejor a medio, corto y largo plazo. Permite crear software de mas calidad.

En IIJ click derecho aparece la opción REFACTOR.

Ejemplos de Refactoring.

* Si tengo una estructura con demasiados if seguidos, podría crear un método que reduzca ese código.
* Si tengo IF else repetidos los saco para no tenerlos duplicados. (con otra función por ejemplo).
* Si tengo atributos de un objeto que son demasiados los puedo agrupar en otra clase.
* Si tengo en un método un código que podría reemplazarse con otro método, tengo la opción con botón derecho refactor – Extract Method.
* Si tengo muchas variables en una operación, haciéndose complicada la lectura de la misma, puedo desglosar en operaciones mas chicas, explicando en cada paso que es lo que voy haciendo.

REFACTORING EN JAVA

Se busca un código con menos deuda técnica. Mas fácil de mantener, entender, escalar.

Es un conjunto de técnicas

Que busca reducir la complejidad..

La deuda técnica se produce cuando tenemos un código que realiza su función pero de una manera muy compleja. Lo que hace que sea cada vez mas costoso mantenerlo en el tiempo.

Con el refactoring mejoramos el código refactorizándolo.

Consolidate Condicional:

Se extrae un if anidado a un método dónde se achica el código con and y or

Código Viejo:

package consolidatedconditional.bad;

public class Main {

private double extraSalary;

private int seniority;

private int education;

private int incidents;

private boolean certification;

double calculateExtraSalary(){

double result = 0;

if (seniority<1){

return 0;

}

if (education<1){

return 0;

}

if(incidents > 10){

return 0;

}

if (!certification){

return 0;

}

// Calculate extra salary:

// ...

return result;

}

}

Refactorizado :

package consolidatedconditional.good;

public class Main {

private double extraSalary;

private int seniority;

private int education;

private int incidents;

private boolean certification;

double calculateExtraSalary(){

double result = 0;

if(!isEligibleExtraSalary()){

return result;

}

// Calculamos el salario extra:

// ...

return result;

}

public boolean isEligibleExtraSalary(){

// agrupar condiciones utilizando AND (&&) y OR (||)

boolean estudios = seniority<1 || education<1;

boolean antiguedad = incidents > 10 && certification;

return estudios || antiguedad;

}

}

CONSOLIDATE DUPLICATE Sacar lo que se repite en un doble if fuera dela estructura de los mismos.

En éste caso hay un código que se repite.

package consolidateduplicate.bad;

public class Main {

private double extraSalary;

private int seniority;

private int education;

private int incidents;

private boolean certification;

double calculateExtraSalary(){

double result = 0;

if(isEligibleExtraSalary()){

result = 500;

sendMessage(); // metodo que se repite en la condicion

}else{

result = 0;

sendMessage(); // metodo que se repite en la condicion

}

return result;

}

private void sendMessage() {

// send email

// connect smtp

System.out.println("Sending message");

}

private boolean isEligibleExtraSalary() {

boolean firstLevel = seniority<1 || education<1;

boolean secondLevel = incidents > 10 || !certification;

return firstLevel || secondLevel;

}

}

Código mejorado: Lo que se hace es sacar lo que se repite de la estructura del IF

package consolidateduplicate.good;

public class Main {

private double extraSalary;

private int seniority;

private int education;

private int incidents;

private boolean certification;

double calculateExtraSalary(){

double result = 0;

if(isEligibleExtraSalary()){ // verdadero

result = 500;

}else{ // faso

result = 0;

}

sendMessage();

return result;

}

private void sendMessage() {

// send email

// connect smtp

System.out.println("Sending message");

}

private boolean isEligibleExtraSalary() {

boolean firstLevel = seniority<1 || education<1;

boolean secondLevel = incidents > 10 || !certification;

return firstLevel || secondLevel;

}

}

Técnica extcact class: Si tenemos una clase grande que tiene demasiados atributos podemos crear otra clase que pertenezaca a la anterior.

package extractclass.bad;

public class Person {

private Long id;

private String firstName;

private String lastName;

private Integer age;

// address

private String street;

private String postalCode;

private String city;

private String country;

private String door;

}

Codigo Nuevo:

|  |  |
| --- | --- |
| package extractclass.good;  public class Person {  private Long id;  private String firstName;  private String lastName;  private Integer age;  private Address address;    } | package extractclass.good;  public class Address {  // address  private String street;  private String postalCode;  private String city;  private String country;  private String door;  } |

EXTRACT MÉTODO:

En el ejemplo se ve que en el código hay dos métodos y luego 4 líneas de código que pueden agruparse en un método de manera que quede mas homogéneo.

[java\_basico/sesiones\_4\_5\_6/ob-refactor/src at main · Open-Bootcamp/java\_basico · GitHub](https://github.com/Open-Bootcamp/java_basico/tree/main/sesiones_4_5_6/ob-refactor/src)

|  |  |
| --- | --- |
| package extractmethod.bad;  public class Main {  void printHTML(){    printHead();  printBody();  System.out.println("This is the footer");  System.out.println("script");  System.out.println("copyright");  System.out.println("");    }  private void printBody() {  }  private void printHead() {  }  } | package extractmethod.good;  public class Main {  void printHTML(){    printHead();  printBody();  printFooter();  }  private void printFooter() {  System.out.println("This is the footer");  System.out.println("script");  System.out.println("copyright");  System.out.println("");  }  private void printBody() {  }  private void printHead() {  }  } |

EXTRACT VARIABLE

En éste caso hay un cálculo complejo entre variables y se hace mas sencillo

|  |  |
| --- | --- |
| package extractvariable.bad;  public class Main {    void printProductPrice(Order order){  Double totalPrice = order.getPrice() \* order.getQuantity() -  order.getOffer() + order.getShipping() \* 2;    System.out.println(totalPrice);  }  } | package extractvariable.good;  import extractvariable.bad.Order;  public class Main {    void printProductPrice(Order order){  // calcular precio total  // 1. precio de los productos  Double quantityPrice = order.getPrice() \* order.getQuantity();  // 2. descuento  Double offerPrice = order.getPrice() \* order.getOffer();  // 3. calcular precio productos con el descuento incluido  Double finalPrice = quantityPrice - offerPrice;  // 4. precio envio  Double shippingCost = 0.0;  if(finalPrice < 50){  shippingCost = 2.99;  }  // precio definitivo  System.out.println(finalPrice + shippingCost);  }  } |

TECNICA DE MAGIC NUMBERS

Cuando tenemos un número metido en el código, ejemplo el costo de envío, podría pasar que lo tenga an varios lugares. La idea es reeemplazarlo por una constante global. Desde el entorno nos permite crear constante.

Las constantes se escriben en mayúsculas separadas por guin bajo.

Ejempo: PRECIO\_MINIMO\_APLICABLE =100. Llevan el atributo FINAL.

|  |  |
| --- | --- |
| package magicnumbers.bad;  public class Main {  public static void main(String[] args) {  double price = 129.99;  double discountPrice = calculateDiscount(price);  double shippingPrice = calculateShipping(price);  }  private static double calculateShipping(double price) {  return price < 100 ? 4.99 : 0; // hard coded    }  private static double calculateDiscount(double price) {  return price \* 0.1;  }  } | package magicnumbers.good;  public class Main {  private static final double PRECIO\_MINIMO\_COMPRA = 100;  private static final double PRECIO\_ENVIO = 2.99;  private static final double ENVIO\_GRATIS = 0;  private static final double DESCUENTO = 0.1;  public static void main(String[] args) {  double price = 129.99;  double discountPrice = calculateDiscount(price);  double shippingPrice = calculateShipping(price);  }  private static double calculateShipping(double price) {  return price < PRECIO\_MINIMO\_COMPRA ? PRECIO\_ENVIO : ENVIO\_GRATIS; // hard coded    }  private static double calculateDiscount(double price) {  return price \* DESCUENTO;  }  } |

return price < PRECIO\_MINIMO\_COMPRA ? PRECIO\_ENVIO : ENVIO\_GRATIS; // hard coded

La línea superior es como el IF. Si se cumple la condición devuelve precio envío, si no , devuelve ENVIo\_GRATIS

TECNICA REMOVE ASIGN

Para crear funciones puras sse recomienda que los parámetros no se modifiquen.

En el primer caso tenemos una función impura

Lo que se recomienda es que si en una función envío valores no los sobreescriba, haga copias con otras variables y devuelva el valor de sichas variables. En éste caso tendremos una Funcion pura.

La función suma es una función pura porque devuelve la suma de los dos valores, y esos valores no se han modificado.

La programación defensiva consiste en no creer que todos los datos que nos pasan están bien.

Por eso se verifica con un if que no sea un dato null

|  |  |
| --- | --- |
| package removeassign.bad;  public class Main {  double calculateDiscount(Order order, double totalPrice){  if (order == null || order.getPrice() == null) // programacion defensiva  return totalPrice;  if (order.getPrice() > 100)  totalPrice += order.getPrice() \* order.getOffer(); // sobreescribe totalPrice  return totalPrice;  }  } | package removeassign.good;  import removeassign.bad.Order;  public class Main {  double calculateDiscount(Order order, double totalPrice){  double result = totalPrice;  if (order == null || order.getPrice() == null) // programacion defensiva  return result;  if (order.getPrice() > 100)  result += order.getPrice() \* order.getOffer(); // sobreescribe totalPrice  return result;  }  } |

Técnica Replace method

|  |  |
| --- | --- |
| public class Product {  double price;  int quantity;  double discount;  double shipping;  // constructores  // metodos  // tostring  public double calculatePrice(){  double result = 0;  double priceQuantity = this.price \* this.quantity;  double priceDiscount = this.price \* this.discount;  double priceShipping = shipping \* 0.50;  // .......  // ......  return result;  }    // getter y setter  public double getPrice() {  return price;  }  public int getQuantity() {  return quantity;  }  public double getDiscount() {  return discount;  }  public double getShipping() {  return shipping;  }  public void setPrice(double price) {  this.price = price;  }  public void setQuantity(int quantity) {  this.quantity = quantity;  }  public void setDiscount(double discount) {  this.discount = discount;  }  public void setShipping(double shipping) {  this.shipping = shipping;  }    } | package replacemethod.good;  /\*\*  \* Técnica de refactoring para componer métodos  \* (Composing Methods)  \*  \* Replace method with method object  \*/  public class Product {    private double price;  private int quantity;  private double discount;  private double shipping;      // metodos especiales    double calculatePrice(){  // ProductPriceCalculator calculator = new ProductPriceCalculator(this);  // return calculator.calculatePrice();    return new ProductPriceCalculator(this).calculatePrice();  }    // getter y setter  public double getPrice() {  return price;  }  public int getQuantity() {  return quantity;  }  public double getDiscount() {  return discount;  }  public double getShipping() {  return shipping;  }  public void setPrice(double price) {  this.price = price;  }  public void setQuantity(int quantity) {  this.quantity = quantity;  }  public void setDiscount(double discount) {  this.discount = discount;  }  public void setShipping(double shipping) {  this.shipping = shipping;  }      }  package replacemethod.good;  public class ProductPriceCalculator {  private double price;  private int quantity;    public ProductPriceCalculator(Product product) {  this.price = product.getPrice();  this.quantity = product.getQuantity();  }      public double calculatePrice(){  double result = 0;    // calculos      return result;  }  } |

Métodos de la Clase String

package com.sergio;  
  
public class StringsCadenas {  
 public static void main(String[] args) {  
  
 //METODOS DE LA CLASE STRING//  
 //length devuelve largo de la cadena de texto.  
  
 String mensaje = " Hola mundo ";  
 System.*out*.println(mensaje.length());  
   
 String mensajeMAY = mensaje.toUpperCase(); //Devuelve el texto en mayúscula  
 System.*out*.println(mensajeMAY);  
 // para guardar el texto convertido debo hacer lo siguiente:  
 mensajeMAY = mensajeMAY.trim();  
  
 System.*out*.println(mensajeMAY.trim()); // Corta los esppacios en blanco antes y después de la cadena.  
  
 String otro = "HOLA MUNDO";  
 if (mensajeMAY.equals(otro)){ //funcion Equals: permite comparar si una cadena es igual a otra.  
 System.*out*.println("verdadero");  
 }  
 String a = "BOQUITA";  
 String b = "boquita";  
 if (a.equalsIgnoreCase(b)){ //IgnoreCase: permite comparar ignorando mayúsculas y minúsculas  
 System.*out*.println("verdadero");  
 }  
  
  
 }  
}

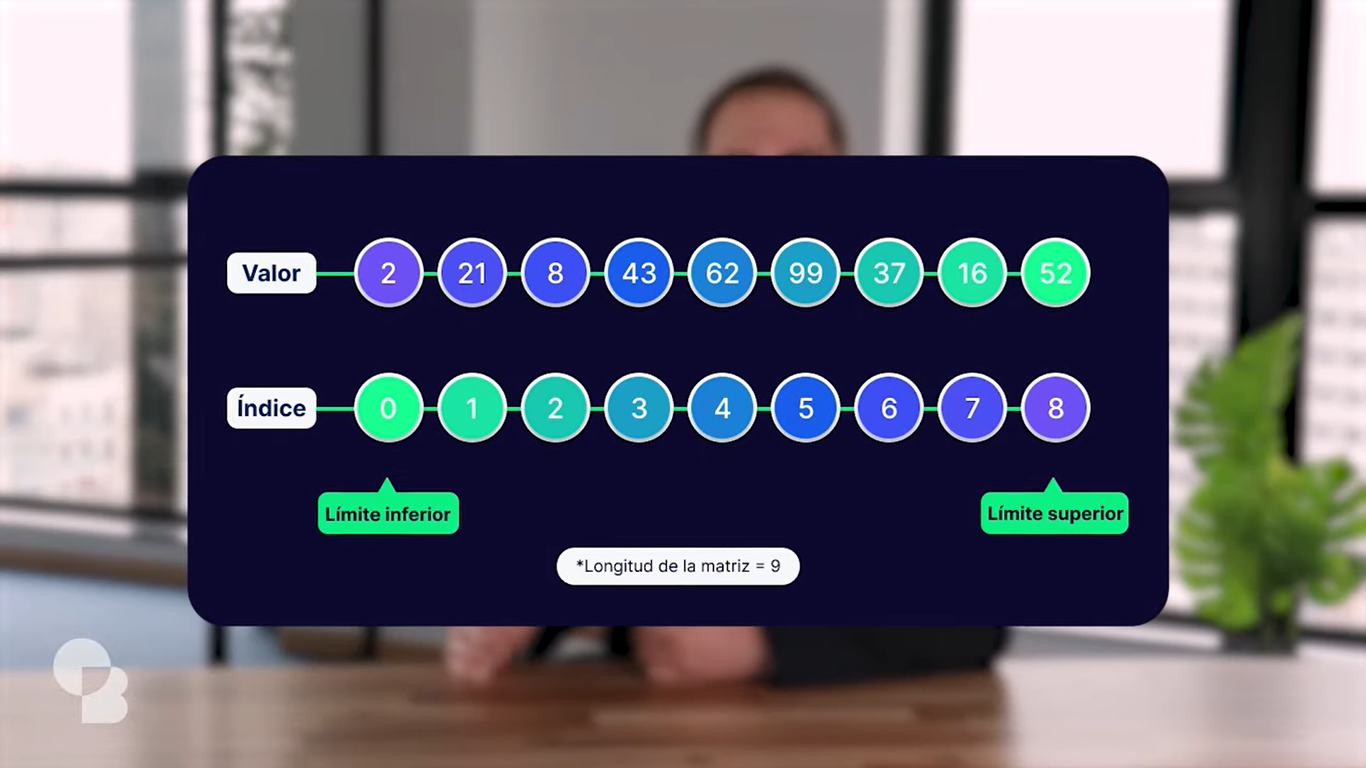
ARRAYS

Un array es un vector de variables.

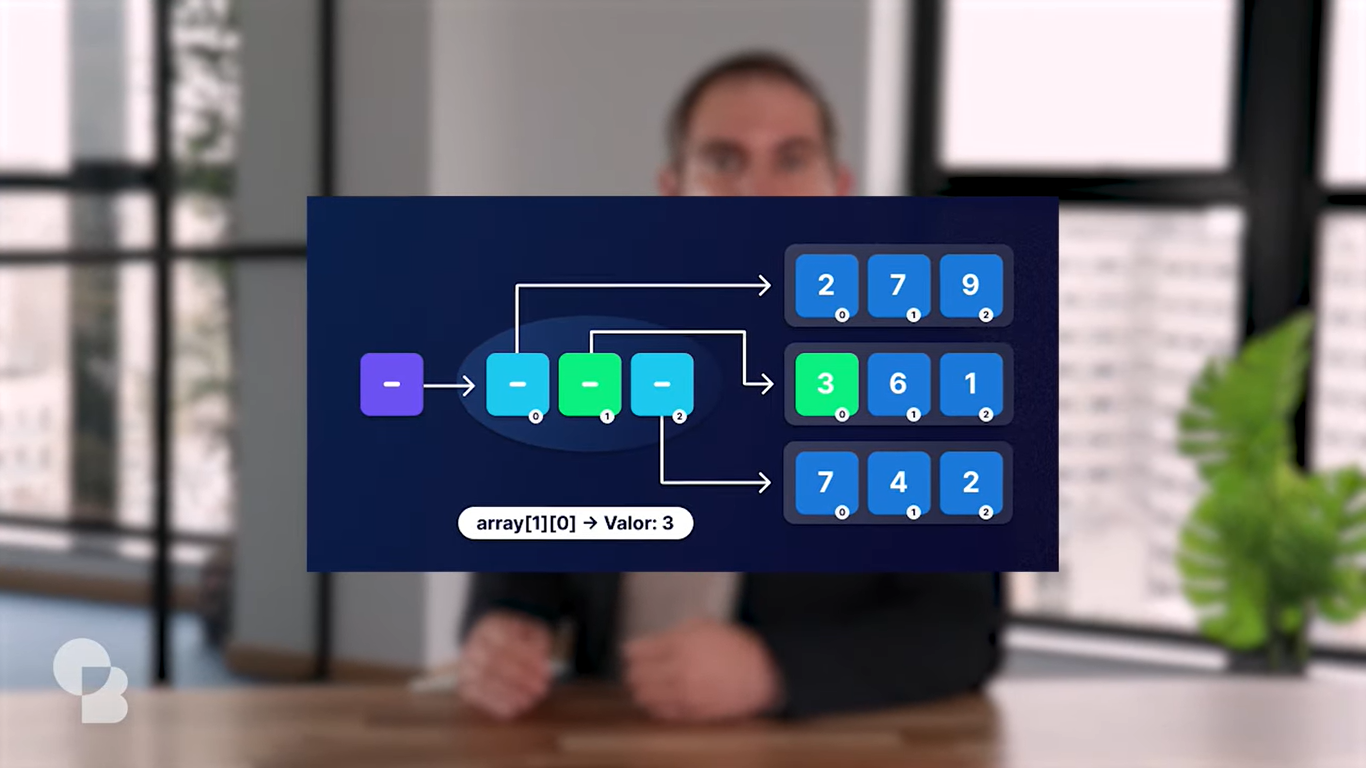
Podemos agrupar una serie de variables en un vector denominado Array.

El Array se compondrá de un orden y un valor para ese orden.

ARRAY UNIDIMENSIONAL:



ARRAY de mas de una dimensión.



package com.sergio;  
  
public class arrayClase1 {  
  
 public static void main(String[] args) {  
 // Generaremos un array de números enteros. Por ejemplo para las notas de un alumno.  
 int[] notas = new int[10]; // De ésta manera indicamos que creamos un arry que va a contener 10 notas  
  
 int[] notas2 = {8, 10, 5, 9, 8, 7, 5, 6}; //Si se las notas lo creo de ésta otra forma.  
  
 notas[0] = 8;  
 notas[1] = 9;  
 System.*out*.println(notas[0]);  
 // Una forma de recorrer el Array:  
 for (int i = 0; i < notas.length; i++) { //notas.lenght nos da el largo del array  
 System.*out*.println(notas2[i]);  
 }  
 //otra forma de recorrer el array  
 for (int item:notas2) {  
 System.*out*.println(item);  
 }  
  
 // Métodos de Arrays muy utilizados:  
 // binary search: DEVUELVE LA POSICIÓN EN LA QUE SE ENCUENTRA UN ELEMENTO  
 notas2.binarySearch(notas2,9);  
   
 // SORT: PERMITE ORDENAR UN ARRAY   
 notas2.sort(notas2);  
 System.*out*.println(notas2);  
   
 //MÉTODO EQUALS, COMPARA 2 ARRAY  
 equals(notas, notas2);  
   
   
 }  
   
}

LISTAS DINÁMICAS.

Son parecidas a los arrays, pero nos permiten agregar y borrar elementos dinámicamente.

Usamos la Interface **List.**

**List es una interface que nos ofrece una lista de métodos como:**

**Size: Tamaño de una lista.**

**Contains: Para comprobar la existencia de un elemento dentro de la lista.**

**Add: Agregar Elementos.**

**Remove: Remover elementos.**

**Para Crear un Array List:**

List<String> nombre = new ArrayList<>();

**List<**tipo>nombre = new ArrayLisst<>();

package com.sergio;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class listasEnJava {  
 @Override  
 public String toString() {  
 return "listasEnJava{}";  
 }  
  
 public static void main(String[] args) {  
 List<String> nombres = new ArrayList<>(); // Creo la lista  
 nombres.add("Nombre1"); // ADD agrega elementos  
 nombres.add("Nombre2");  
 nombres.add("Nombre3");  
 nombres.add("Nombre4");  
  
 System.*out*.println(nombres);  
  
 for (String nombre : nombres) { // Esta sintaxis permite recorrer el Array guardando en la variable temporal nombre  
  
 System.*out*.println(nombre);  
  
 }  
 //Podemos trasladar éste enfoque a los objetos.  
  
 List<Cocheb> coches = new ArrayList<>();  
 coches.add(new Cocheb("Honda"));  
 coches.add(new Cocheb("Fiat"));  
 coches.add(new Cocheb("Ford"));  
 System.*out*.println(coches);  
 for (Cocheb cocheb : coches) {  
 System.*out*.println(cocheb);  
 }  
  
  
 }  
}

package com.sergio;  
  
  
class Cocheb {  
 String marca;  
  
 public Cocheb(String marca) {  
 this.marca = marca;  
  
 }  
  
 @Override  
 public String toString() {  
 return "Cocheb{" +  
 "marca='" + marca + '\'' +  
 '}';  
 }  
}

Mapas:

Son estructuras de datos que nos permiten trabajar con pares de **clave – valor**.

Usaremos la interfaz Map

package com.sergio;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class maps {  
 public static void main(String[] args) {  
 Map<String, String> personas = new HashMap<>();  
 //Map <tipo de dato, Tipo de dato> Nombre new (Implementacion de Mapa a utilizar)  
 //Hashmap nos va a permitir construir el ejemplo de personas.  
  
 personas.put("24969593", "Sergio"); //Con put agregamos personas  
 personas.put("27019285", "Mari");  
 personas.put("12345678H", "Persona1");  
 personas.put("12345678P", "Persona2");  
  
 System.*out*.println(personas);  
  
 for (String key : personas.keySet()) { //keyset nos devuelve el conjunto de claves  
 System.*out*.println(key);  
 }  
  
 for (String value : personas.values()) { //values nos devuelve el conjunto de nombres(valores)  
 System.*out*.println(value);  
 }  
  
 for (Map.Entry <String,String> pair: personas.entrySet()){ //Esta estructura nos devuelve el par de valores de cada caso  
   
 System.*out*.println(pair.getKey() + " " + pair.getValue());  
 }  
 }  
}

MECANISMOS DE GESTIÓN DE EXCEPCIONES JAVA. TRY CATCH FINALLY

Estos mecanismos los utilizamos cuando tenemos una estructura que podría llegar a provocar un error. Nos permite “Capturar el error” y seguir con la ejecución del programa. Impedimos el programa se detenga.

La idea es crear **Software Robusto**, es decir que no se detenga ante un error en el software.

En el bloque try {} tengo entre llaves la expresión que puede ser problemática.

En el boque catch {} El códigoque queremos llevar a cabo si ocurre ese problema.

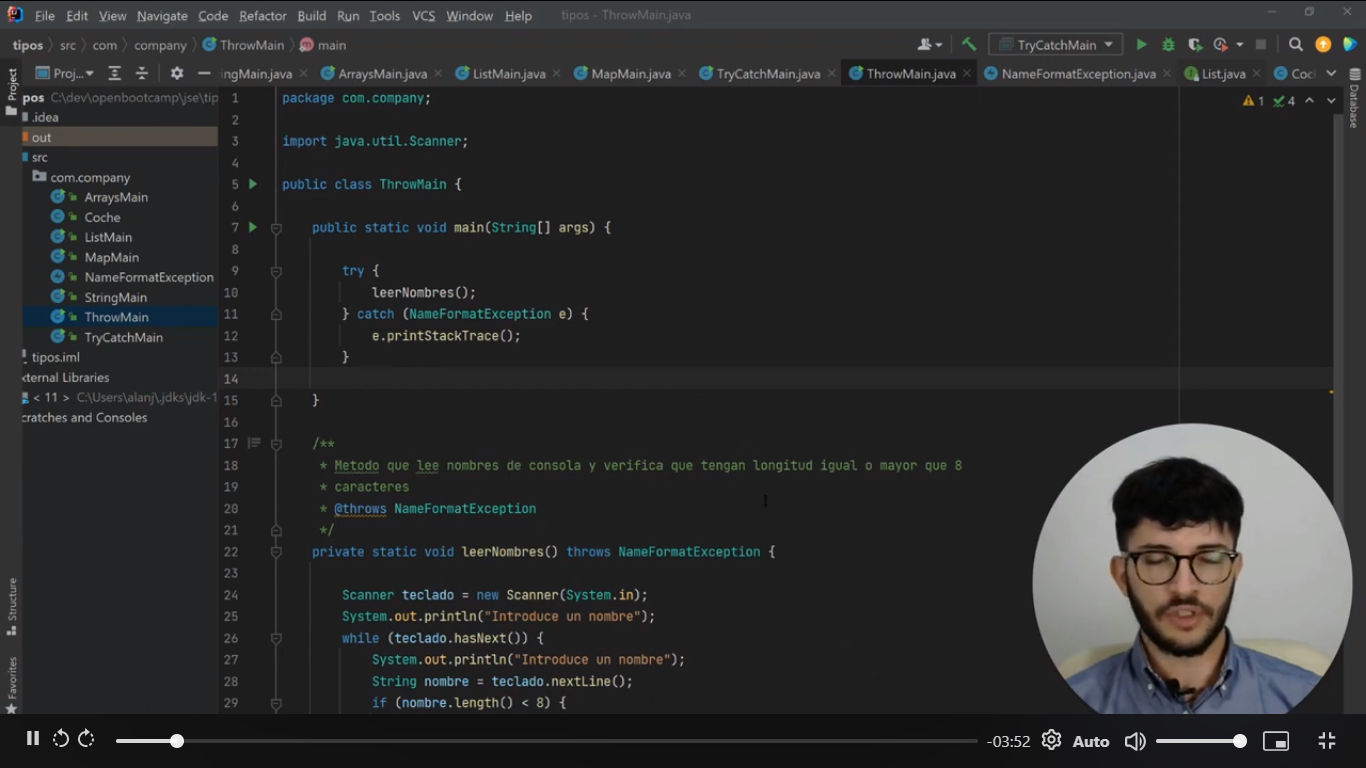
Finally {cerrar recursos}.

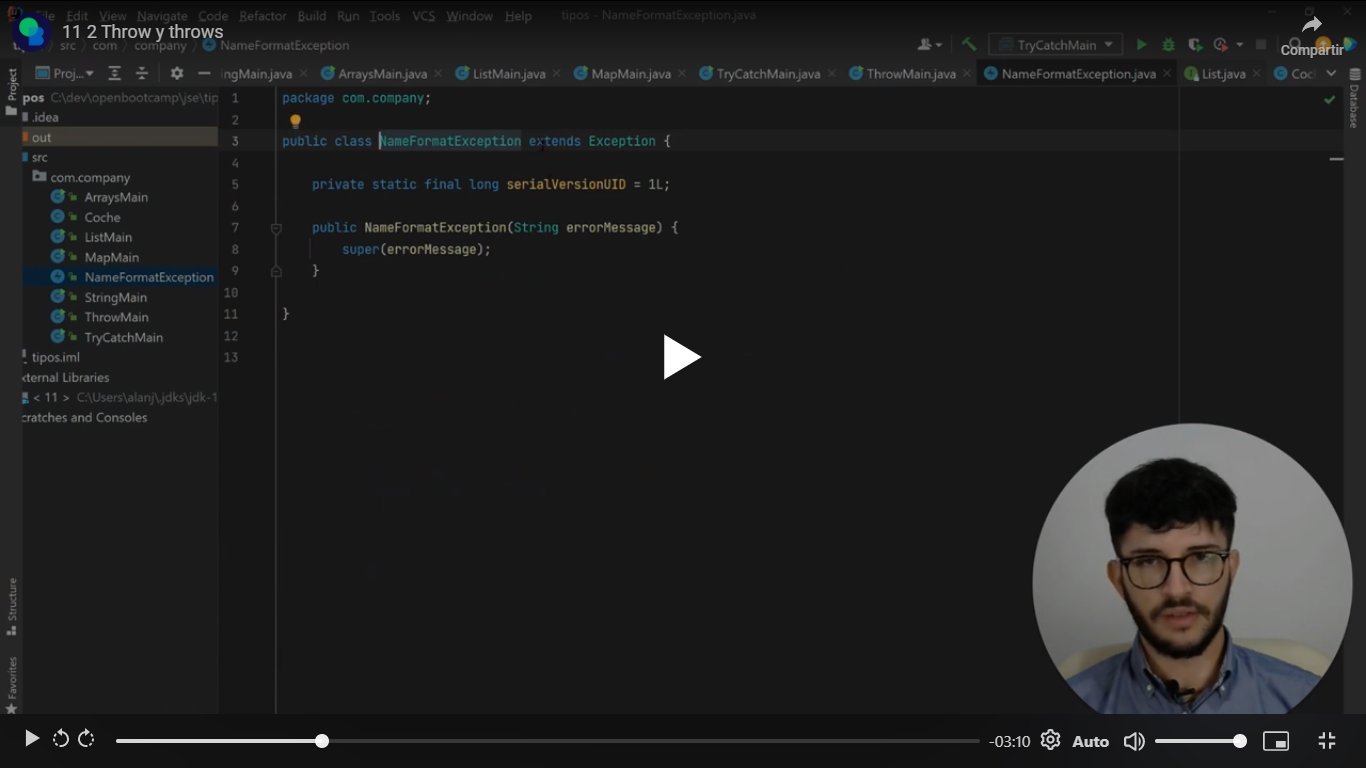
package com.sergio;  
  
public class TryCatch {  
 public static void main(String[] args) {  
  
  
 try {  
 int result = 5/0;  
 } catch (ArithmeticException e) {  
 e.printStackTrace(); // Si anulo ésta línea no veo el error en la consola.  
 }finally {  
 System.*out*.println("Cierre de recursos");  
 }  
 System.*out*.println("Fin del programa");  
 }  
  
}

Throw y throws

throws es en la signatura del método para indicar que lanza una excepción.

Throw para crear la excepción y lanzarla.





TIPOS DE DATOS COMPLEJOS

CADENAS DE Texto:

Se puede declarar:

Var cadena = “Mensaje de texto” En éste caso el compilador define el tipo de variable.

O

String cadena =”Mensaje de texto”.

Operaciones con cadenas de texto String

package datosComplejos;  
  
import java.util.Locale;  
  
public class Cadenas {  
  
 /\*  
 Operaciones con cadenas  
 lenght() devuelve el largo.  
 toLowerCase() Devuelve el valor en Minúsculas.  
 toUpperCase() Devuelve el valor en Mayyúsculas.  
 startsWith() Devuelve true si comienza con el parámetro indicado.  
 endsWith() Compara si termina la cadena con el valor indicado.  
 charAt devuelve el caracter de una posición detreminada. (Recordar que 0 es la primera posición.  
  
 \*/  
 public static void main(String[] args) {  
 String cadena = "Mensaje de TEXTO";  
  
 int cadenaLen = cadena.length(); //Longitud de cadena.  
 System.*out*.println("La longitud de la cadena es " + cadenaLen);  
  
 String cadenaMinus = cadena.toLowerCase(); //Pasa cadena a minúscula  
 System.*out*.println("La cadena ahora es" + cadenaMinus);  
  
 String cadenaMayus = cadena.toUpperCase(); //Pasa cadena a Mayyúscula  
 System.*out*.println("La cadena ahora es" + cadenaMayus);  
  
  
  
 boolean resultado = cadena.startsWith("Men"); //Compara el inicio de la cadena a ver si empieza con Men  
 // si es así devuelve true y lo guardo en resultado.  
 if (resultado) {  
 System.*out*.println("Empieza pos lo que busco");  
 } else System.*out*.println("No empieza por lo que busco");  
  
 boolean resultado2 = cadena.endsWith("to"); //Compara el final de la cadena a ver si termina con to  
 // si es así devuelve true y lo guardo en resultado.  
 if (resultado2) {  
 System.*out*.println("termina por lo que busco");  
 } else System.*out*.println("No termina por lo que busco");  
  
  
 char letra = cadena.charAt(0);  
 System.*out*.println("Caracter es la letra" + letra);  
  
 }  
}

public class cadenas2 {  
 public static void main(String[] args) {  
 String cadena = "Mensaje de texto";  
 //123456789012345  
  
 //Para el ejercicio length, charAT y concatenacion  
 //Recorrer una cadena  
  
 for (int i =0; i< cadena.length();i++){  
 System.*out*.println("Caracter actual: " + cadena.charAt(i));  
  
  
 }  
 }  
}

ARRAYS

En java es una colección de elementos de un mismo tipo de datos. En Java, es una variable que tiene múltiples valores consecutivos con el mismo tipo.

Puedo tener arrays de enteros, String, etc

Para declararlo;

nt []arrayUno = new int[5]

Tipo de Dato[] nombre del Array = new tipo de dato [cantidad de elementos];

Si lo declaro de 5 enteros no puede tener mas.

Otra forma de declararlo e inicializarlo:

Int arrayUno[] = {1,2,3,4,5}

**//Longitud de un Array:**

System.*out*.println("Longitud del array " + arrayUno.length);

// Declarar un array multidimensional   
int [][]arrayBidiUno = new int[2][4]; (filas, columnas)crear

En un array puedo mutar los valores

Ejemplo:

String nombres[] = {“Pepe”,”Juan”};

nombres[0] =”Sergio”;

Cambié le valor 0 del array. Ahora es Sergio y no Victor.

Los Arrays no pueden acrecentarse ni decrecer luego de ser creados.

package datosComplejos.arrays;  
  
public class demoArrays {  
 public static void main(String[] args) {  
  
  
 // Declarar un array  
 int []arrayUno = new int[5];  
  
 // Asignar valores a un array previamente declarado:  
 arrayUno[0] = 1;  
 arrayUno[1] = 2;  
 arrayUno[2] = 3;  
 arrayUno[3] = 4;  
 arrayUno[4] = 5;  
  
 // Declarar e inicializar un array  
 String []arrayDos = {"Naranja", "Limon", "Pomelo"};  
  
  
 //Longitud de un Array  
 System.*out*.println("Longitud del array " + arrayUno.length);  
 // Recorrer un array:  
 for (int i = 0; i < arrayUno.length; i++) {  
 System.*out*.println("Posicion del array " + i + " contiene el valor " + arrayUno[i]);  
 }  
  
 // Recorrer un array con la forma corta: En algunos círculos la llaman for each  
 for (String citrico : arrayDos) {  
 System.*out*.println("Cítrico actual " + citrico);  
 }  
  
 // Declarar un array multidimensional  
 int [][]arrayBidiUno = new int[2][4];  
 /\*Podría crearlo e inicializarlo si conociera sus valores de la siguiente manera:  
  
 int arrayBidi[][] = {  
 {1,2,3,4]  
 {10,20,30,40}  
 }  
  
 \*/  
  
 // Asignar valores a un array multidimensional previamente declarado:  
 arrayBidiUno[0][0] = 1;  
 arrayBidiUno[0][1] = 2;  
 arrayBidiUno[0][2] = 3;  
 arrayBidiUno[0][3] = 4;  
  
 arrayBidiUno[1][0] = 10;  
 arrayBidiUno[1][1] = 20;  
 arrayBidiUno[1][2] = 30;  
 arrayBidiUno[1][3] = 40;  
  
 // Recorrer un array bidimensional:  
  
 //El primer For es para las filas.  
 //El segundo for es para las columnas.  
 for (int i = 0; i < arrayBidiUno.length; i++) {  
 for (int j = 0; j < arrayBidiUno[i].length; j++) {  
 System.*out*.println("Posición: [" + i + "][" + j + "] y contiene el valor " + arrayBidiUno[i][j]);  
 }  
 }  
  
 // Declarar e inicializar un array multidimensional  
 int [][]arrayBidiDos = {  
 {1, 2, 3, 4},  
 {10, 20, 30, 40},  
 };  
  
 // Recorrer un array bidimensional:  
 for (int i = 0; i < arrayBidiDos.length; i++) {  
 for (int j = 0; j < arrayBidiDos[i].length; j++) {  
 System.*out*.println("Posición: [" + i + "][" + j + "] y contiene el valor " + arrayBidiDos[i][j]);  
 }  
 }  
  
 // "Crecer" un array.  
 // Solo podemos hacerlo cre ando uno nuevo de más tamaño y llenándolo con el contenido del original.  
 int nuevoArrayLen = arrayUno.length \* 2;  
 int []nuevoArray = new int[nuevoArrayLen];

VECTORES

Son similares a los array pero no cecesitan tamaño inicial ni final si no queremos.

CREACIÓN;

Vector <Integer> vector = new Vector;

**Vector**<*tipo de dato*> *nombre del vector* = **new Vector**

Cuando creamos un vector, en realidad por debajo creamos un array de una capacidad determinado. Cuando voy agregando elementos voy llenando ese array de 10 elementos. El tamaño es la cantidad de datos que introducí y la capacidad es el tamaño del array subyacente.

Si me paso de la capacidad del array automáticamente aumenta creando por debajo otro array.

Por eso un vector es un array dinámico.

package datosComplejos;  
  
import java.util.Vector;  
public class vectores{  
public static void main(String[]args) {  
  
  
// Un vector es similar a un array, pero puede (de)crecer dinámicamente.  
 Vector<Integer> vector = new Vector<>();  
  
 // Añadir elementos  
 vector.add(1);  
 vector.add(2);  
 vector.add(3);  
 System.*out*.println("Contenido del vector: " + vector);  
  
 // Quitar un elemento:  
 vector.remove(1);  
 System.*out*.println("Contenido del vector: " + vector);  
  
 // Los vectores tienen capacidad y tamaño:  
 System.*out*.println("Tamaño: " + vector.size() + " - Capacidad: " + vector.capacity());  
  
 // Podemos ajustar la capacidad al tamaño actual:  
 vector.trimToSize();  
 System.*out*.println("Tamaño: " + vector.size() + " - Capacidad: " + vector.capacity());  
  
 // Al añadir un elemento al vector, si superamos su capacidad, esta aumenta en capacityIncrement.  
 // capacityIncrement se declara en el constructor del vector, y si es menor o igual a cero, por  
 // defecto es el doble de la capacidad previa.  
 vector.add(9);  
 System.*out*.println("Tamaño: " + vector.size() + " - Capacidad: " + vector.capacity());  
  
 // Recorrer un vector  
 for (int i = 0; i < vector.size(); i++) {  
 System.*out*.println("Valor en la posicion " + i + " del vector: " + vector.get(i));  
 }  
  
 // Recorrerlo con la forma corta:  
 for (int i : vector) {  
 System.*out*.println("Valor actual del vector: " + i);  
 }  
  
 // Comparar dos vectores  
 Vector<Integer> vector2 = new Vector<>();  
 vector2.add(1);  
 vector2.add(3);  
 vector2.add(9);  
  
 if (vector.equals(vector2)) {  
 System.*out*.println("Los dos vectores son equivalentes");  
 System.*out*.println(" -> vector: " + vector);  
 System.*out*.println(" -> vector: " + vector2);  
 }  
  
 //Trabajando varios conceptos:  
  
for (int i=0;i< vector.size() ; i++){  
 if (i % 2 == 0) {  
 vector.remove(i);  
 continue;  
 /\* EL continue se pone porque cuando borra el vector de la posición "i" en la línea de abajo  
 vector.get(i) estoy tratando de acceder a un elemento borrado. El continue indica que pase  
 a la siguiente iteracción del for y no llega a esa línea.  
 \*/  
 }  
 System.*out*.println("Valor es " + vector.get(i) + "en posición " + i);  
}  
  
/\*  
En el siguiente ejemplo se ve claramente el fallo si no anulo la 3ra línea.  
Se estaría tratando de acceder a la posición borrada en la línea anterior. Eso da un FALLO!  
 \*/  
 System.*out*.println(vector.get(1));  
 vector.remove((1));  
 //System.out.println(vector.get(1));  
  
  
 //El vector 2 es de capacidad 10 (por defecto) y tamaño 3 (son los elementos).  
 //Se puede consirmar con la siguiente línea  
 System.*out*.println("Capacidad" + vector2.capacity() + "tamaño "+ vector2.size());  
  
 //Para decrecer el tamaño del vector :  
 vector2.trimToSize(); //Reduce la capacidad del vector hasta su tamaño  
 System.*out*.println("Capacidad" + vector2.capacity() + "tamaño "+ vector2.size());  
  
 }  
}

ARRAYLIST

Es la implementación de la interfaz “list”. Es muy similar a un vector.

La capacidad del Array es automática. Por eso no puedo usar lista.capacity como en los vectores.

Cuando se llena el Araylist incrementa su capacidad en un 50%.

Los Arraylist no están sincronizados por lo tanto no puedo modificar en forma concurrente (paralela) un mismo arraylist. Es inseguro para éste tipo de operaciones.

En la programación secuencial se puede trabajar con Arraylist, en cambio en la concurrente o multihilo trabajaremos con vectores.

¿Podemos convertir una lista en un array? Si, se puede un tipo complejo como es el arraylist en otro tipo complejo como es el array

LinkedList: Tiene los mismos métodos y propiedades. Se declara de la misma manera.

Opera de la misma forma.

¿Cuándo usaremos una u otra?. Depende de nuestro programa.

Diferencias entre ambas listas al pie del ejemplo.

package datosComplejos.arrayList;  
  
import java.util.ArrayList;  
import java.util.LinkedList;  
  
public class arrayList {  
 public static void main(String[] args) {  
 // ArrayList: Una lista en formato array.Implementan la interfaz "list"  
 // Similares a los vectores, usando un array de forma subyacente.  
 // No están sincronizados, es importante tenerlo en cuenta cuando necesitamos código "thread-safe".  
  
 // Tambien varía la forma de aumentar su capacidad. Por defecto, el vector dobla la capacidad, y  
 // el ArrayList aumenta con la formula "capacidad += capacidad \* 0.5". En el constructor no podemos  
 // darle un tamaño de autocrecimiento (a diferencia del Vector), y utiliza la formula previa.  
  
 // Crear una lista de tipo ArrayList:  
  
 ArrayList<String> lista = new ArrayList<>();  
  
 // Añadir elementos  
 lista.add("Uno");  
 lista.add("Dos");  
 lista.add("Tres");  
 System.*out*.println("Contenido de la lista: " + lista);  
  
 // Quitar un elemento: Se puede hacer referencia a su posición  
 //lista.remove(1) o si es un String también especificando el contenido como en el siguiente caso:  
  
 lista.remove("Dos");  
 System.*out*.println("Contenido de la lista: " + lista)  
  
 // Las listas tienen tamaño:  
 System.*out*.println("Tamaño: " + lista.size());  
  
 // Recorrer una lista  
 for (int i = 0; i < lista.size(); i++) {  
 System.*out*.println("Valor en la posicion " + i + " de la lista: " + lista.get(i));  
 }  
  
 // Recorrerlo con la forma corta:  
 for (String e : lista) {  
 System.*out*.println("Valor actual de la lista: " + e);  
 }  
  
 // Comparar dos listas  
 ArrayList<String> lista2 = new ArrayList<>();  
 lista2.add("Uno");  
 lista2.add("Tres");  
  
 if (lista.equals(lista2)) {  
 System.*out*.println("Las dos listas son equivalentes");  
 System.*out*.println(" -> lista: " + lista);  
 System.*out*.println(" -> lista2: " + lista2);  
 }  
  
 // Podemos convertir una lista en un array:  
 String []array = new String[lista.size()];  
 for (int i = 0; i < lista.size(); i++) {  
 array[i] = lista.get(i);  
 }  
  
 for (String elemento : array) {  
 System.*out*.println("Elemento de la lista convertida en array es: " + elemento);  
 }  
  
 // Otra forma de hacer lo mismo:  
 for (Object arrayObjeto : lista.toArray()) {  
 System.*out*.println("Elemento de la lista convertida en array es: " + arrayObjeto.toString());  
 }  
  
 // Las listas pueden ser de más tipos, además de ArrayList:  
 LinkedList<String> listaEnlazada = new LinkedList<>();  
  
 // Y tienen las mismas operaciones que un ArrayList:  
 listaEnlazada.add("Hola");  
 System.*out*.println(listaEnlazada.get(0));  
 listaEnlazada.remove("Hola");  
  
 // Y pueden copiarse unas a otras  
 LinkedList<String> listaEnlazadaDos = new LinkedList<>(lista);  
  
 // Podemos recorrer nuestra nueva lista, tipo enlazada, y a la que hemos copiado  
 // los valores desde "lista", que es del tipo ArrayList.  
 for (String elemento : listaEnlazadaDos) {  
 System.*out*.println("Elemento actual en la lista enlazada: " + elemento);  
 }  
  
 // Cada tipo de lista tiene sus pros y sus contras:  
 // ArrayList:  
 // - Utiliza un array dinámico internamente  
 // - Es más rápida para almacenar y buscar datos  
 // - Solo implementa la interfaz "List", por lo que solo puede actuar como una lista  
 //  
 // LinkedList:  
 // - Utiliza una lista doblemente enlazada a nivel interno  
 // - Es más rápida para modificar datos  
 // - Puede funcionar como lista y como cola, ya que implementa las interfaces "List" y "Deque"  
 //  
 // Hay más tipos, ¡búscalos!  
 }  
 }

BIG DECIMAL

Todos los lenguajes de programación tiene problema con la precisión en grandes volúmenes de datos. Por ejemplo muestra que system.out,println (0,1f+0,1f+0,1f……….(10 veces), debería dar uno y muestra en pantalla 1,0000001

No usar ni DOUBLE ni FLOAT para datos FINANCIEROS.

En Java lo que requiere precisión se implementa con la clase BIGDECIMAL.

package datosComplejos.bigDecimal;  
Recordar que :

Big Decimal no usa operadores tradicionales (+,-,\*,/, etc.). en su lugar add, multiply, etc.

La sumo o resto a un valor el segundo y el resultado queda en el primero.

import java.math.BigDecimal;  
  
public class bigDecimalejemplo {  
 public static void main(String[] args) {  
  
// Math es muy preciso con decimales utilizando BigDecimal.  
 double valorInicial = 3.14f;  
 BigDecimal valorA = BigDecimal.*valueOf*(valorInicial);  
  
 double valorSecundario = 3.15f;  
 BigDecimal valorB = BigDecimal.*valueOf*(valorSecundario);  
  
 // Suma el BigDecimal "valorB" a "valorA"  
 BigDecimal resultado = valorA.add(valorB);  
  
 // Imprime la suma  
 System.*out*.println(resultado);  
  
 // Se utilizan operadores diferntes:  
  
 double factor1 =4;  
 BigDecimal factor1Big = BigDecimal.*valueOf*(factor1);  
 BigDecimal resultado2 = valorA.multiply(valorB);  
  
 }  
  
 }

MAPAS

Los mapas reciben varios nombres. Los mas comunes son Mapas y Array asociativo.

Es una especie de diccionario donde tengo una clave y un valor. Ese valor podría ser otro mapa, podría ser un bigdecimal, un int, un Array, etc.

La forma mas primitiva de Mapa es el HashMap

En los hasmap no puedo tener la clave duplicada.

Los declaro indicando los 2 tipos de valores que voy a tener, por ejemplo, <String, Integer>

Put, para agregar elemento.

Si repito la clave, el valor se sobreescribe.

package datosComplejos.mapas;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class mapas {  
 public static void main(String[] args) {  
  
  
 // Los mapas implementan colecciones "clave" = "valor".  
  
 // Crear un mapa del tipo "HashMap":  
 HashMap<String, Integer> mapa = new HashMap<>();  
  
 // Añadir elementos:  
 mapa.put("clave1", 10);  
 mapa.put("clave2", 20);  
 mapa.put("clave3", 30);  
  
 // Imprimir un valor de una clave del mapa accediendo a él:  
 System.*out*.println("Valor de \"clave2\": " + mapa.get("clave2"));  
  
 // Eliminar una clave del mapa:  
 mapa.remove("clave2");  
  
 // Un mapa no puede tener claves duplicadas, esto no se añadirá de nuevo, pero si reemplazará  
 // su valor por "4":  
 mapa.put("clave3", 4);  
 System.*out*.println("Valor de \"clave3\": " + mapa.get("clave3"));  
  
 // Un mapa puede cambiar el valor de una clave. Es más rapido reemplazar una clave que conozcamos  
 // con replace que mediante mapa.put pero si la clave no existe no crea el valor(No hace nada):  
 mapa.replace("clave3", 100);  
  
 // Recorrer un mapa:  
 for (Map.Entry<String, Integer> elemento : mapa.entrySet()) {  
 System.*out*.println("Clave: " + elemento.getKey() + " - Valor: " + elemento.getValue());  
 }  
 }  
}

// Recorrer un mapa:  
for (Map.Entry<String, Integer> elemento : mapa.entrySet()) {  
 System.*out*.println("Clave: " + elemento.getKey() + " - Valor: " + elemento.getValue());

Map.Entry : es un tipo de dato. Por lo tanto para acceder con el for declaro una variable temporal del tipo Map.Entry que la llamo elemento. A dicha variable le asignamos el entryset().

Elemento: es clave y valor

Para acceder a la clave getkey()

Para acceder al valor getValue()

El TREE Map es otro tipo de mapa. Se usa igual, la implementación subyacente es lo que cambia.

GESTIÓN DE ERRORES.

Un error es un comportamiento no esperado en nuestro programa.

Los errores se dividen en 3 categorías:

1-ERROR EN TIEMPO DE EJECUCIÓN: Es una situaciónque ocurre en nuestro programa para la cual no estábamos preparadosy que hace que el programa aborte. Ejemplo: System.out.println (5/0). No marca el error el IntJ. Hasta que ejecuto.

En el ejemplo se ve un código dónde se pide 2 números para una división al usuario.

Si introduce 0 como se produce el error en tiempo de ejecución.

2-ERROR DE COMPILACIÓN:

Ejemplo me falta un punto y coma y llamo al compilador. Es un error sintáctico.

3-ERROR LÓGICO O HUMANO :

Lo que he indicado al programa que haga está mal dicho.

Son los mas difíciles de detectar. Por ejemplo poner en un if que si la temperatura en mayor a 25º ponga “Hace frío”.

Avisos:

Warnings

EL compilador me advierte por ejemplo que he importado cosas que no estoy utiliando.

En java podemos suprimir los warnings con @SupressWarnings.

Se recomienda siempre resolverlos.

GESTION DE ERRORES:

TRY \_ CATCH \_ FINALLY

Cuando se produce un error en tiempo de ejecución se dice que se produce una excepción.

Cuando quiero capturat posibles excepciones de mi código, tengo que utilizar una construcción especial del lenguaje que se llama try, catch y opcionalmente finality.

Dentro de try escribo entre llaves el código que quiero proteger. El que puede dar error.

Dentro del catchpuedo poner la excepción que quiero capturar o todas.

Cuando yo no se que excepción se produce puedo capturarlas todas. Para esto utulizo }

catch (Exception e) {} donde Exception es un tipo de dato ya definido y la variable se denomina e por convención.

En e se guarda la excepción.

Si yo quiero capturar una excepción específica se indica. Por ejemplo:

Catch (ArithmeticExcepcion e).

Finally. Se ejecuta pase lo que pase (Haya error o no). Hay controversia en su utilización.

package gestioErrores;  
import java.util.Scanner;  
  
  
public class Errores {  
 public static void main(String[] args) {  
  
  
 //Capturando todas las excepciones.  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.print("Introduce dos números: ");  
  
 int numeroA = scanner.nextInt();  
 int numeroB = scanner.nextInt();  
 try {  
 int resultado = numeroA / numeroB;  
 System.*out*.println("El resultado es: " + resultado);  
 } catch (Exception e) {  
 // } catch (Exception e) { "Si quisiera capturar todas las excepciones. Excepcion es un tipo de dato  
 // y se denomina e la variable por convención.  
  
 System.*out*.println("¡Boom! Excepción es " + e.getClass());  
 }  
  
 //Capturando excepciones concretas:  
 Scanner scanner2 = new Scanner(System.*in*);  
 System.*out*.print("Introduce dos números: ");  
  
 numeroA = scanner2.nextInt();  
 numeroB = scanner2.nextInt();  
 try {  
 int resultado = numeroA / numeroB;  
 System.*out*.println("El resultado es: " + resultado);  
 } catch (ArithmeticException e) {  
  
 System.*out*.println("¡Boom! Excepción es " + e.getClass());  
 } catch (Exception e ){  
 System.*out*.println("Estoy en una excepción que no es aritmética" + e);  
 }finally {  
 System.*out*.println("Finally");  
 }  
  
//finally es opcional. Se puede usar para cerrar recursos.  
  
  
 }}

GENERAR EXCEPCIONES.

Throw (singular): Se utiliza para generar una excepción.

Throws(plural): Se utiliza para declarar si una función devuelve excepciones.

package gestioErrores;  
  
import java.io.IOException;  
  
public class ThrowThrows {  
 public static void main(String[] args) {  
 try {  
 *divide*(4, 0);  
 }catch (Exception e) {  
 System.*out*.println("Excepcion es " + e);  
 }  
 }  
 public static double divide (int A, int B) throws ArithmeticException, IOException {  
 int resultado =0;  
 try {  
 resultado = A / B;  
 } catch (ArithmeticException e){   
 throw new IOException(); // Acá genero una excepción IOExeption  
 }  
 return resultado;  
 }  
  
 /\*  
 Mi función divide "puede" generar una excepción aritmética (la division por 0).  
 Como programador puedo ocuparme del código con un try catch, o puedo indicar que la función puede  
 fallar. Con throws lo indico. La función debe invocarse en un bloque try catch  
 \*/  
}

ENTRADA Y SALIDA

INPUT STREAM

InputStream es una clase que nos devuelve un flujo de caracteres (Un flujo de bytes).

Cuando hablamos de stream es que continuamente estamos recibiendo datos.

Esos datos podemos aceptarlos y leerlos como entrada o descartarlos.

package EntradaSalida;  
import java.io.FileInputStream;  
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.io.InputStream;  
  
public class EntradaSalida   
{  
 public static void main(String[] args) {  
  
  
 try {  
 InputStream fichero = new FileInputStream("C:/Ejercicio 3.txt");  
/\* Arriba declaramos una variable del tipo InputStream, Inicializamos una instancia de FileInputStream.  
Como la clase Input stream puede arrojar error debe invocarse desde un try catch  
 \*/  
  
 try {  
 byte[] datos = fichero.readAllBytes();//Este código lee *todo el fichero. Ojo memoria.* //Como la función FileImputStream nos devuelve un fichero en bytes, declaro la variable datos del tipo byte y  
 //le asigno *todo el fichero con fichero.readAllBytes()* for (byte dato : datos) {  
 System.*out*.print((char) dato);// Si pongo int me da números  
 }  
 // AL usar el tipo char muestra caracteres de fichero y no números (bytes)  
 fichero.close();  
  
 } catch (IOException e) {  
 System.*out*.println("No puedo leerlo: " + e.getMessage());  
 }  
  
  
 } catch (FileNotFoundException e) {  
 System.*out*.println("Fichero no encontrado: " + e.getMessage());  
 }  
 *leerByteAByte*();  
 }  
  
  
 public static void leerByteAByte() {  
 try {  
 InputStream fichero = new FileInputStream("C:/Ejercicio 3.txt");  
  
 try {  
 byte []datos = new byte[5];  
 int data = fichero.read(datos);  
  
//EL código de abaqjo lee el archivo byte a byte, es mas lento pero no tengo problemas de memoria  
 while (data != -1) {  
 //Cuando devuelve -1 es cuando llego al final del fichero  
 System.*out*.println("Leido: '" + (int)data + "' que es el código " + data);  
 data = fichero.read();  
 }  
  
 } catch (IOException e) {  
 System.*out*.println("Error al leer: " + e.getMessage());  
 }  
  
 } catch (FileNotFoundException e) {  
 System.*out*.println("Error al abrir fichero: " + e.getMessage());  
 }  
 }}

Dice que no hace falta saber todas éstas formas porque está la opción –buffer Stream

BUFFER\_ Input:\_STREAM

No trabaja con ficheros. Trabaja con entradas.Lo va leyendo de a poco y lo va devolviendo

CLASE SCANNER

La clase scanner nos permite solicitar datos del usuario.

Cuando hacemos un system.in , automáticamente se genera un inputstring.

package EntradaSalida;  
  
import java.util.InputMismatchException;  
import java.util.Scanner;  
  
public class scanner {  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 boolean ok = false;  
 do {  
 System.*out*.println("Ingrese dos números");  
 scanner.reset();  
 try {  
 int a = scanner.nextInt();  
 int b = scanner.nextInt();  
 ok = true;  
 } catch (InputMismatchException e) {  
 System.*out*.println("Números invalidos.......");  
 }  
  
 } while (!ok);  
  
 }  
 }

Print Stream

Hay varios tipos.

El mas común es el que hemos usado de salida.

PrintStream escribe en un fichero

public class InputStram {  
 public static void main(String[] args){  
//Leer y copiar un fichero  
  
 //Lee Fivhero  
 try {  
 InputStream in = new FileInputStream("C:/Ejercicio 3.txt");  
 byte[] datos = in.readAllBytes();  
 in.close();//buena práctica, cerrar fichero origen  
 //Escribe fichero  
 PrintStream out =new PrintStream("C:/destino.txt");  
 out.write(datos);  
 out.close();// Buena práctica: Cerrar archivo destino   
 } catch (Exception e) {  
 System.*out*.println("Excepción " + e.getMessage());  
 }  
 }  
}

Eventos XML y JSON

Eventos:

Un evento es algo que se produce entre el emisor y el receptor o receptores.

Eventos en Java:

En java nos referimos a eventos en una “Interfaz de Usuario”. Mover una ventana es un evento. Seleccionar texto es un evento, etc.

Con Swing podemos implementar una interfaz de usuario, es decir lo que el usuario ve.

Event Handle:

Es el código que se dispara cuando se produce el evento.

Me posiciono en el paquete. Clic en New. Luego Swing UI Designer. GUI Form

Creando Eventos:

JFrame frame = new JFrame("Mi aplicación"); //Declaro frame del tipo JFrame  
frame.setContentPane(new MainForm().taxPane);//Instancio  
frame.setVisible(true); //Hago visible la aplicacion con el método setVisible

Create Listener

Un listener va a estar en espera de que un evento se prcoduzca.

Me paro sobre el botón- Botón secundario del mouse. – Create Listener – Accion listener.

Action listener

Se puede generar de manera automática.

[Sesion 10 · Open-Bootcamp/java\_basico@e134672 (github.com)](https://github.com/Open-Bootcamp/java_basico/commit/e134672957f941d5045e7a79ff3b6e1125dddedd)EVENTOS

|  |
| --- |
| package eventos; |
|  |  |  |
|  |  | import java.util.ArrayList; |
|  |  |  |
|  |  | interface Mensajero { |
|  |  | void hanSaludado(); |
|  |  | } |
|  |  |  |
|  |  | class Receptor implements Mensajero { |
|  |  | @Override |
|  |  | public void hanSaludado() { |
|  |  | System.out.println(Math.random()); |
|  |  | } |
|  |  | } |
|  |  |  |
|  |  | class ReceptorMusical implements Mensajero { |
|  |  | @Override |
|  |  | public void hanSaludado() { |
|  |  | System.out.println("Soy un receptor musical"); |
|  |  | } |
|  |  | } |
|  |  |  |
|  |  | class Emisor { |
|  |  | private ArrayList<Mensajero> receptores = new ArrayList<Mensajero>(); |
|  |  |  |
|  |  | public void meteReceptor(Mensajero receptor) { |
|  |  | receptores.add(receptor); |
|  |  | } |
|  |  |  |
|  |  | public void saluda() { |
|  |  | for (Mensajero ml : receptores) { |
|  |  | ml.hanSaludado(); |
|  |  | } |
|  |  | } |
|  |  | } |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | public class Main |
|  |  | { |
|  |  | public static void main(String []args) { |
|  |  | Emisor emisor = new Emisor(); |
|  |  | Receptor receptor = new Receptor(); |
|  |  | Receptor receptor2 = new Receptor(); |
|  |  | Receptor receptor3 = new Receptor(); |
|  |  | Receptor receptor4 = new Receptor(); |
|  |  | ReceptorMusical rm = new ReceptorMusical(); |
|  |  |  |
|  |  | emisor.meteReceptor(receptor); |
|  |  | emisor.meteReceptor(receptor2); |
|  |  | emisor.meteReceptor(receptor3); |
|  |  | emisor.meteReceptor(receptor4); |
|  |  | emisor.meteReceptor(rm); |
|  |  |  |
|  |  | emisor.saluda(); |
|  |  | } |
|  |  | } |

[Sesion 10 · Open-Bootcamp/java\_basico@e134672 (github.com)](https://github.com/Open-Bootcamp/java_basico/commit/e134672957f941d5045e7a79ff3b6e1125dddedd)