**REPASO CLASE 3. Metodo parsebyte(String s, int radix):**

El metodo Byte.parseByte(String s, int radix) analiza el string pasado como parámetro y lo considera como un nro byte en la base que especifiquemos en el segundo parámetro radix. Nuestro sistema decimal utiliza una base 10. Por ejemplo:

System.out.println("Parse String to byte: "+Byte.parseByte("100", 10));

Nos muestra un 100.

Pero podemos usar un sistema hexadecimal

System.out.println("Parse String to byte with radix octal: "+ Byte.parseByte("123", 8));

Porque se calcula el nro 123 en base 8: 1 \* 8^2 + 2 \* 8 + 3 = 64 + 16 + 3 = 83

**CLASE 4 – Aspectos Avanzados de OOP**

***Polimorfismo con vinculación dinámica***

**Upcasting**

Es el procedimiento para transformar un objeto de una clase a otra siempre y cuando haya una relación de herencia entre ambas.

Se produce a nivel objetos. Ej: La clase Ejecutivo hereda de la Clase Empleado (Clase Padre).

// Java me permite cambiar el Modificador con el cual estamos viendo al objeto “Ejecutivo”. Es decir me permite cambiar:

**Ejecutivo** ej1 = new Ejecutivo(("Máximo Dueño" , 2000);

por

**Empleado** *e1* = new **Ejecutivo**("Máximo Dueño" , 2000); //**Principio de Sustitucion**: Podemos usar un objeto de la subclase cuando el programa espere un objeto de la superclase.

Referencia = creo el objeto

Esta operación en donde un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases se la denomina Upcasting.

Cuando hagamos este tipo de casteos hay que tener cuidado porque para la referencia e1 no existen los atributos y métodos de la clase Ejecutivo, aunque la referencia apunte a un objeto de este tipo.

Si ahora existe una variable de tipo Empleado y se desea acceder a los métodos de la clase derivada - suponiendo que contiene un método ejecutarPlanificacion() en la clase Ejecutivo -, hay que convertir explícitamente la referencia de un tipo a otro.

Esto se hace con el operador de cast de la siguiente forma:

Empleado =>

// Un objeto creado con el constructor de Ejecutivo esta referenciando a la clase empleado.

Empleado emp = new Ejecutivo("Máximo Dueño" , 2000);

Si quiero recuperar un atributo de Ejecutivo, creo un puntero de ejecutivo y lo apunto hacia emp.

Ejecutivo ej = emp. (java me pregunta ¿estamos seguros que dentro de emp hay un ejecutivo?.

Entonces casteamos.

// Se convierte (o castea) la referencia de tipo Ejecutivo

Ejecutivo ej = (Ejecutivo) emp; **=> convierte la referencia de tipo Empleado asignándola a una**

**Referencia de tipo Ejecutivo.** Para el compilador es correcto

Porque Ejecutivo es una clase derivada de Empleado. En

Tiempo de ejecución la JVM convertirá la referencia.

Caso contrario, lanzara una excepción.

Una vez que casteamos me salen todos los métodos relacionados a Ejecutivo.

Vemos el Proyecto “Clase 4” donde tenemos las clases Ejecutivo, Empleado y el main.

**Netbeans:**



**Eclipse:**



**CLASE ABSTRACTA**

LEEMOS CLASE ABSTRACTA DEL DOC “Clase 4 – POO Avanzado”



Consideraciones:

- un método abstracto solo puede estar en una clase abstracta.

- un método estatico puede existir en una clase abstracta.

- Cuando se declara una clase hija de una abstracta hay que implementar los métodos abstractos **si o si**. No es necesario implementar los otros métodos.

- Solo en la definición de los métodos abstractos puedo no poner un cuerpo del método. Para el resto de los métodos (estáticos y de instancia) si se requiere especificar la implementacion.

**Resumen conceptos claves:**

* Concepto abstracto.
* No puede ni debe ser instanciado.
* Base.
* Clases hijas heredan atributos y métodos.
* Metodos abstractos solo implementados en subclases. { }

**Netbeans:**



**Eclipse:**

****

Las clases abstractas solo pueden implementar los métodos de acceso publico. Pueden tener métodos normales o abstractos.

Esto se relacionado mucho con el concepto de **Interfaz**.

**INTERFACES – Parte 2**

Es un contrato entre un concepto y una implementación.

Se aplica para clases que están distribuidas en el sistema que no necesariamente esten relacionadas. Pero necesitan tener un comportamiento similar en cierto aspecto. Por ejemplo una intefaz facturación donde se implemente para diferentes clases que requieran facturar.

Es una declaracion de comportamiento, es decir un conjunto de métodos sin su implementación. Aunque puedo definir atributos que terminan siendo constantes.

Define un protocolo de comportamiento. Una interfaz puede heredar de otras interfaces.

**Definición de una interfaz**

public interface Imprimible{

public void imprimir();

public void imprimir(Tamano t, Estilo s);

}

**Utilización**

Ejemplo en Java, utilización

public class Documento **implements** Imprimible{

public void imprimir(){

// Esto es un comentario. Aqui va la definicion.

}

public void imprimir(Tamano t, Estilo s){

// Esto es un comentario. Aqui va la definicion.

}

**Codigo Netbeans:**



**Codigo Ecipse:**

****

**Notas:**

En la interfaz solo se indica los nombres de los métodos, sus parámetros y tipos que retornan, pero no el código de cada método.

Un método estatico o de instancia dentro de una clase podría recibir como parámetro un objeto de tipo interfaz. Que se representa como una de las clases que implementa esa interfaz.

**Laboratorio Integrador (Ejercitacion Adicional, no esta en el Alumni):**



**Codigo Netbeans**

****

**Codigo Eclipse**

****

**INTERFAZ FUNCIONAL (JAVA 8)**

Es toda aquella interface que tenga solamente un metodo abstracto. Es decir puede implementar uno o mas métodos default, pero deberá tener forzosamente un único método abstracto. Un método abstracto es un método sin implementar.

El concepto de interfaz funcional es nuevo y fue agregado a partir de la versión 8 de Java. Cobra importancia al utilizar expresiones lambda (λ).

1.Creamos una interfaz funcional

package com.mycompany.functionalinterface;

@FunctionalInterface

public interface IStrategy2 {

public String sayHelloTo(String name);

public default String sayHelloWorld(){

return "Hello World";

}

}

Otra forma de asegurarnos de que estamos definiendo correctamente una interface funcional, es anotarla con @FunctionalInterface , ya que al anotarla el IDE automáticamente nos arrojara un error si no cumplimos con las reglas de una interface funcional.

2.**Metodos Default (Java 8)**:

Solo se permiten introducir en Interfaces, por lo tanto no existen en clases. Estos métodos, al igual que todos los demás en las interfaces, son de manera implícita públicos. Su principal diferencia es que no son abstractos como el resto y necesitan proporcionar una implementación para pasar la fase de compilación.

Hasta ahora, cuando implementábamos una interfaz en una clase, esta nos obligaba a implementar también todos sus métodos. Sin embargo no pasa lo mismo con los métodos por default que ya estarán disponibles en esas clases sin haberlos implementado.

Por lo tanto estos métodos añaden nuevas funcionalidades a nuestras interfaces manteniendo la compatibilidad con el código ya existente.

4.Interfaces Funcionales, introducción a expresiones Lambda:

Veamos como se puede utilizar la interfaz IStrategy2 mediante expresiones Lambda.

5.En el **Main** colocamos lo siguiente:

package com.mycompany.functionalinterface;

public class Main {

public static void main(String[] args) {

IStrategy2 strategy = (name) -> "Hello " + name;

System.out.println(strategy.sayHelloTo("Juan Rosales"));

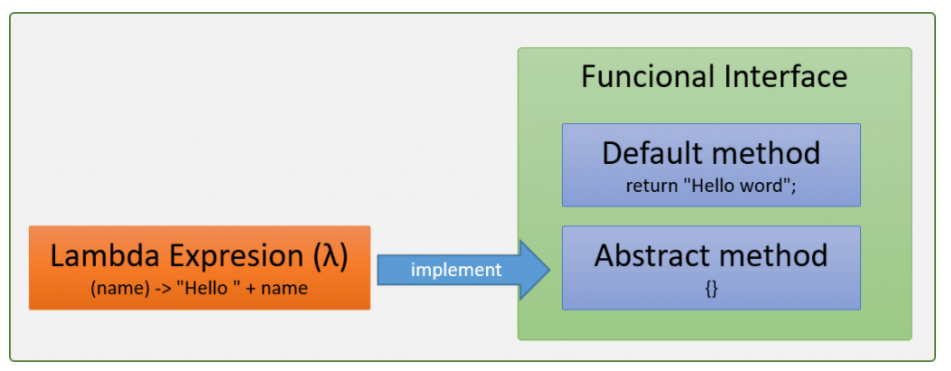
System.out.println(strategy.sayHelloWorld());

}

}

Definimos una expersion Lambda en donde concatena **Hello** con el nombre de la persona pasada como parámetro. Luego esta expresión es asignada a la variable **strategy,** la cual es de tipo **IStrategy**. Luego de esto mandamos llamar al método **sayHelloTo** el cual nos regresa el saludo para “Juan Rosales”. Luego llamamos al método **sayHelloWorld,** el cual manda un “Hello World”.

Como vemos la **expresion Lambda** siempre se asigna al **metodo abstracto,** por este motivo el metodo **sayHelloTo** es implementado con el cuerpo de la expresion Lambda, mientras que el metodo **sayHelloWorld** continua con la misma implementación que viene desde la interface.



Cuando una expresion Lambda es asignada a una interface, esta siempre implementara el metodo abstracto, es por esta razón por la que solo puede existir un metodo abstracto y muchos defaults.

Codigo:

**Netbeans:**

****

**Eclipse:**

****

**PAQUETES**

Paquete es un medio de organización de clases, para agrupar clases e interfaces. Se suelen crear a partir de necesidades funcionales. Pueden contener clases, interfaces y mas paquetes.

**Sintaxis**:

package mipaquete.misubpaquete;

public class Persona{

}

**Utilizacion:**

import mipaquete.misubpaquete.\*;

**KEYWORD FINAL**

Es una palabra reservada de Java. Tiene una semántica distinta según donde se la utilice.

**Atributos** => valor **constante** Ej

public final int valorCuota= 100;

o public static final String HOLA = “Hola”; //es el único caso en que el nombre del atributo va en mayúsculas.

**Metodos** => El método que tiene la palabra clave **final**  , la clase que lo contiene será la ultima en que se podrá implementar el mismo.

Ej:

public final void moverPieza(){

//definición del método

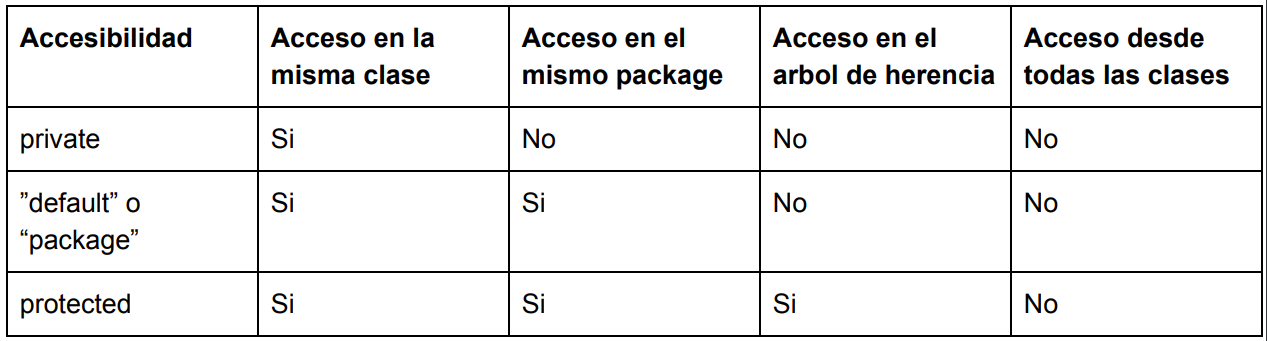
}

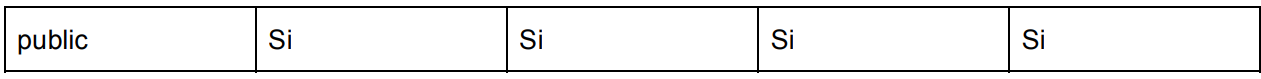
**Clases**=> No se puede extender, no puede generar una subclase que se extienda de ella.

Ej: public final class Hombre{

}

**Accesibilidad: Identificadores**

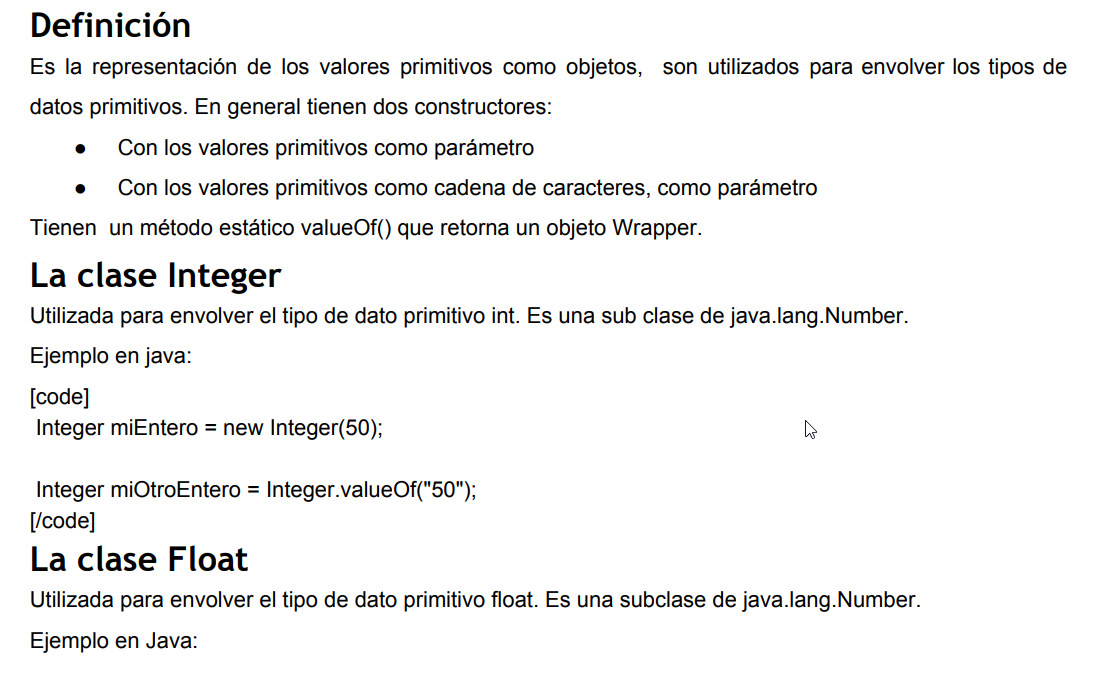
****

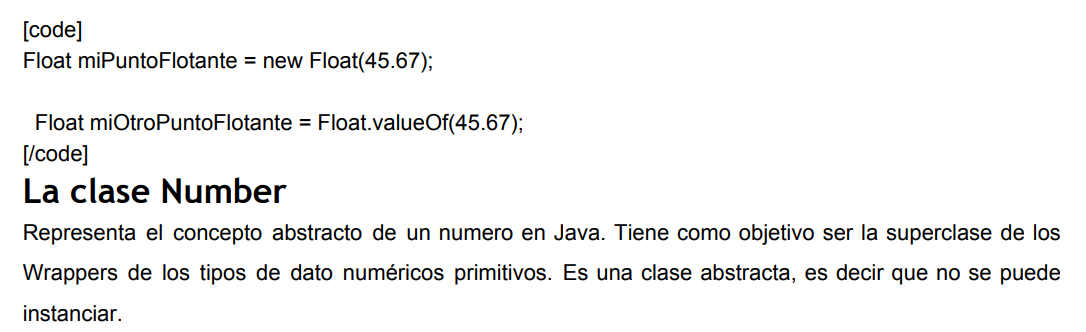
****

**Nota: protected** solo se refiere a atributos y métodos. Para el resto incluye clases, métodos y atributos. Para el caso de **default** la clave esta en el package.

protected string atributo1;

**WRAPPERS de los TIPOS DE DATOS PRIMITIVOS.**



****

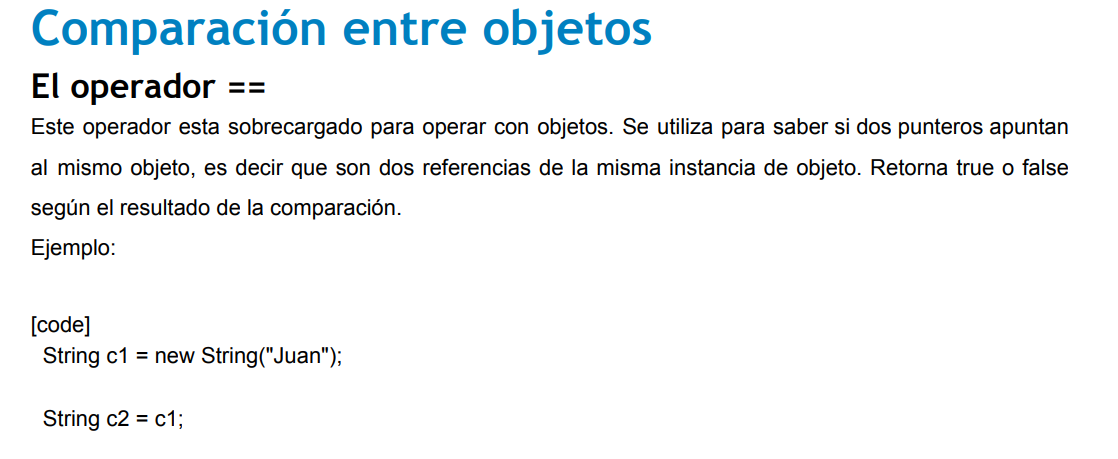
**Ejemplo: Codigo “Clase4-Wrappers”**

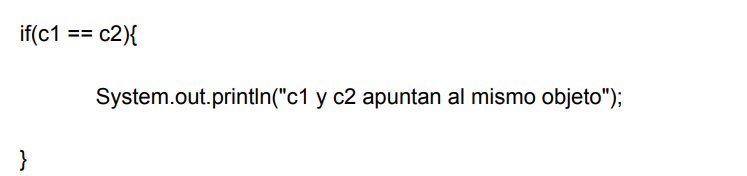
**Netbeans**

****

**Eclipse**

****

****

****

**En este caso apuntan al mismo objeto.**

Pero en el siguiente ejercicio:

String a = "hola";

String b = "hola";

String c = new String("hola");

System.out.println(a==b);

System.out.println(a==c);//porque le pedi a Java que me asigne un nuevo

//lugar en memoria

System.out.println(b==c);//idem previo

//en conclusion el operador == no es de lo mas correcto para comparar Strings. No me compara carácter por carácter. Sino lugares en memoria.

**Uso equals** y vemos como nos da todo true. Porque equals compara contenido. No referencias en memoria.

String a = "hola";

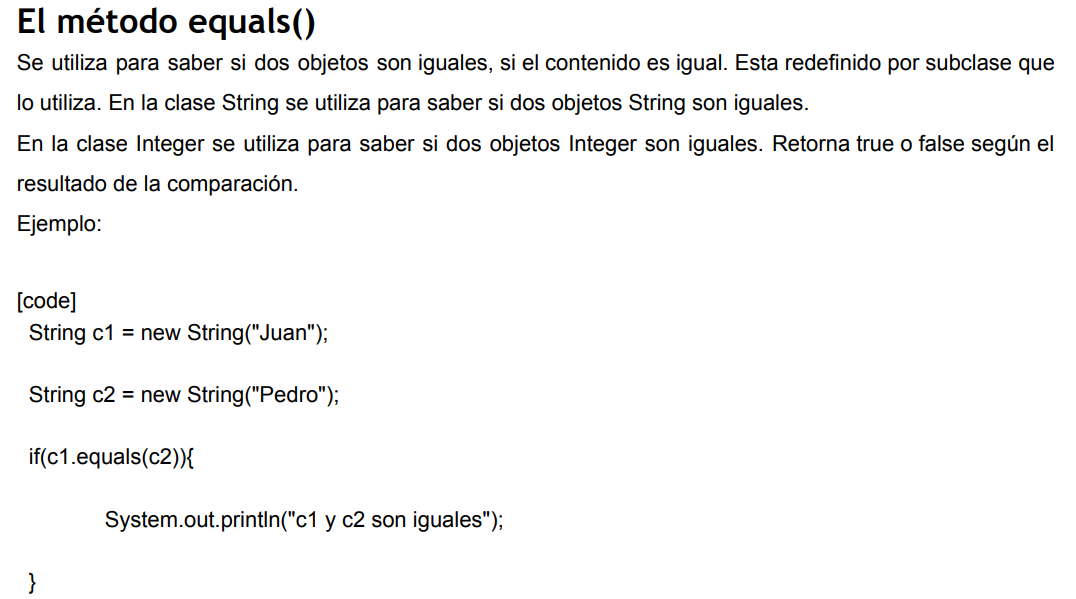
String b = "hola";

String c = new String("hola");

System.out.println(a.equals(b));

System.out.println(a.equals(c));

System.out.println(b.equals(c));



**Ejercicio: Codigo “Clase4-Equals”**

Ejemplo => piensan que esto dará true o false, porque?. Paso los parámetros via set.

Persona p1 = new Persona();

p1.setNombre("Carlos");

Persona p2 = new Persona();

p2.setNombre("Carlos");

System.out.println(p1.equals(p2));

REFERENCIA = PUNTERO.

El método equals esta definido en la clase object el cual hace igual a igual. Lo cual en este caso para la clase objecto comparamos lugares de memoria (referencias) para objetos. Object=>equals()

Para solucionarlo defino que criterio utilizo para considerar un nombre igual a otro.

En nuestro caso es que el nombre sea igual carácter por carácter.

Entonces para ello en la clase Persona (sobreescribimos el método equals):

package clase4.equals;

public class Persona {

private String nombre;

//1. Piso el equals de la clase Object, ponemos nuestra propia inteligencia

@Override

public boolean equals(Object o) {

Persona persona = (Persona)o; //1ro casteamos lo que nos esta enviando el metodo

//para que la referencia apunte a un objeto de tipo persona y no aun objeto de tipo object.

return this.nombre.equals(persona.getNombre()); //identifico los atributos, hace referencia a this. Porque es un objeto local del metodo

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

}

Vuelvo a probar en el main

package clase4.equals;

public class Clase4Equals {

public static void main(String[] args) {

Persona p1 = new Persona();

p1.setNombre("Carlos");

Persona p2 = new Persona();

p2.setNombre("Carlos");

System.out.println(p1.equals(p2));

}

}