Conceptos básicos

Compilación y ejecución

```
Compilación
javac NombreDelArchivo.java
Ejecución
java NombreDelArchivo
Paso de parámetros
java NombreDelArchivo arg1 arg2
```

Tipos de datos

Tipo	Tamaño	Ejemplo	Default
boolean	no determinado	true / false	0
byte	1 byte	128	0
short	2 bytes	32,767	0
char	2 bytes	'a'	'\u0000'
int	4 bytes	212,345	0
float	4 bytes	1234.12F	0.0f
long	8 bytes	123948573L	0
double	8 bytes	1247593.1739	0.0d

Comentarios

```
Comentarios de una
sola línea
// Comentario
Comentarios de multiples
líneas
/*
Comentario
de múltiples
líneas
*/
```

Sentencia if

```
if (expresión booleana) {
      //Acción en caso verdadero
} else {
      //Acción en caso falso
}


if (expresión booleana) {
      //Acción en caso verdadero
} else if (expresión booleana) {
      /**
      * Acción en caso de que la segunda
      condición sea verdadera
       **/
} else {
      /**
      * Acción en caso de que ninguna
      condición se cumpla
       **/
}
```

Operadores aritméticos

```
Suma                a + b
Resta               a - b
Multiplicación      a * b
División            a / b
Módulo              a % b
Incremento          a ++
Decremento          a --
Pre incremento      ++ a
Pre decremento      -- a


Atajos  de asignación

a +=1; Incrementa a en 1

a -=1 Decrementa a en 1
```

Niveles de acceso

```
private      Misma clase

default      Mismo paquete

protected    Diferente paquete a
             través de herencia

public       Desde cualquier
             lugar
```

Creación de objetos

```
[Referencia] nombre=new
Constructor(parámetros);

Ejemplo

Perro roscko=new Perro();
```

Loop while

```
while (expresión booleana) {
      //Sentencias a repetir
}
```

Operadores de comparación

```
menor que       a <  b
mayor que       a >  b
menor o igual   a <= b
mayor o igual   a >= b
Igual           a == b
Diferente       a != b
```

Definición de variables

```
tipo de dato nombre = valor

Ejemplo

int x = 0 ;
```

Operador ternario

```
( expresión booleana )?
valor si es verdadero :
valor si es falso

Ejemplo

boolean permitirPaso = (edad > 18)
? true : false;
```

Loop do while

```
do {
      /*
      * Sentencias a repetir, se ejecuta al
      menos una vez
       */
} while (true);
```

Operadores lógicos

```
a = true    b=false

AND     a && b = false
OR      a || b = true
NOT     a != false

Nota: && y || se les conoce
como de corto circuito de tal
modo que, si con evaluar el
primer elemento es suficiente
para determinar el resultado,
el segundo elemento no se
evalúa.

En caso de que no se desee ese
comportamiento y se quieran
ejecutar ambos, es posible
utilizar & y |.
```

Definición de atributos

```
[modificador de acceso] [tipo de
dato] [nombre de variable] = valor;

ejemplo
public int edad=18;
```

Operador instanceof

```
Devuelve verdadero si a es
una instancia creada a
partir de la clase Perro

boolean res=a instanceof Perro;
```

For loop

```
for (inicialización;
evaluación;incremento / decremento
/ acción) {
      // Sentencias a repetir
}

for (int i = 0; i < 10; i++) {
      // Sentencias a repetir
}

continue: Deja de ejecutar la
iteración actual y salta a la
siguiente.


break: Rompe el ciclo actual.

Es posible utilizar tags para
determinar cual de los dos ciclos
se desea romper.

outer: for (int i = 0; i <= 10; i++) {
            for (int j = 0; j <= 10; j++) {
                  if (i == 5) {
                        break outer;
                  }
            }
      }
```

Sentencia switch

```
switch (e) {
case 'a':
      /*
      * sentencias en caso de que e sea
      igual a 'a'
       */
 break;
case 'b':
      /*
      * sentencias en caso de que e sea
      igual a 'b'
       */
 break;
default:
      /*
      * sentencias en caso de que e no
      sea ninguna de las anteriores
       */
 break;
}
```

Definición de métodos

```
[modificador de acceso] valor de
retorno nombre del método
(parámetros){
    //Sentencias
}

Ejemplo
public String getName(){
  return "Alex";
 }
```

Operadores a nivel de bits

```
Corrimiento a la izquierda
    <<

Corrimiento a la derecha
    >>

Operador AND &

Operador OR  |

Operador XOR ^
```

Definición de constructores

```
[modificador de acceso]  nombre de
la clase ( parámetros){
    //Sentencias
}

Ejemplo

public Perro(String name) {
  System.out.print("Se creó  " + name);
 }
```

For each loop

```
for (int temp : array) {
 // Sentencias a repetir
}
```

Definición de una clase

```
[modificador de acceso] class
NombreDeLaClase {
//Atributos
//Constructores
//Métodos
}

Ejemplo

public class Perro {
 private String nombre;
 public Perro() {
 }
 public void ladrar() {
  System.out.print("WOW");
 }
}
```

Palabra reservada final

```
En una clase : Hace que no
se pueda heredar de ella.

En un método : Hace que no
se pueda sobreescribir.

En una variable primitiva:
No se puede cambiar su
valor.

En una referencia : El
objeto se puede cambiar
pero no se puede apuntar a
otro.
```

DEVS4J
Hecho por Devs para Devs

Programación orientada a objetos

Nombre de la clase

Figura

+ color:String

Atributos

Modificador de acceso:

+ **Public**

- Private

Protected

Métodos

+ calcularArea() : double
- calcularPerimetro() : double

Persona

+ nombre:String
+ fechaDeNacimiento:String

+ dormir() : void
+ respirar() : void
+ comer() : void

Circulo

+ radio:double

+ calcularArea() : double
- calcularPerimetro() : double

Cuadrado

+ lado:double

+ calcularArea() : double
- calcularPerimetro() : double

Estudiante

+ numeroDeCuenta:int

+ aprobar() : void
+ reprobar() : void
+presentarExamen (ExamenNormal e) : float
+presentarExamen (ExamenExtraordinario e) : float

Empleado

+ sueldo:float

+ trabajar() : void
+ cobrar() : void

Coche

- motor:Motor

+avanzar():void

Motor

+encender():void

Examen

- calificacion:float
- numeroDePreguntas: int

+ getCalificacion() : float
+ setCalificacion(float calif) : float

- float sumaAciertos() : float
- float generaPromedio() : float

Herencia

```
Circulo c=new Circulo();
double area=c.calcularArea();

Cada clase que herede de Figura puede crear su propia implementación de los métodos calcularArea() y calcularPerimetro().

Estudiante e=new Estudiante();
e.dormir();
e.respirar();
e.aprobar();

Estudiante hereda de Persona, por esto es posible invocar a los métodos definidos en ella sin tener que re escribirlos.
```

Polimorfismo

```
Figura c=new Circulo();
double area=c.calcularArea();

Es posible asignar un objeto de tipo Circulo a una referencia de tipo Figura gracias al polimorfismo.

Persona e=new Estudiante();
e.dormir();
e.respirar();

Es posible asignar un objeto de tipo Estuidiante a una referencia de tipo Persona gracias al polimorfismo.
```

IS A

Para poder determinar si una clase hereda de otra debe pasar el test ***IS A*** (ES UN), veamos algunos ejemplos:

Perro is an Animal -**Cumple**

Libro is a Cuaderno -**No cumple**

Coche is a Vehiculo -**Cumple**

Si cumple la relación significa que podemos utilizar herencia entre las dos clases, en caso contrario no podemos hacerlo.

HAS A

HAS A (Tiene un) es una relación también conocida como **composición** e indica que una clase tiene como atributo a un objeto de otra clase el cual puede utilizar para invocar a sus métodos, por ejemplo la clase **Coche** tiene un **Motor** y puede invocar a su método encender.

Encapsulamiento

El encapsulamiento permite ocultar parte de nuestro código y exponer solo lo que deseamos, en la clase **Examen** podemos acceder al valor de la calificación, pero solo a través de métodos ***getter*** y ***setter***.

```
Examen e=new Examen();
float calificacion = e.getCalificacion();
```

Si se quiere acceder directamente a la calificación sin utilizar el método getCalificación del siguiente modo:

```
Examen e=new Examen();
e.calificacion;
```

Obtenremos el siguiente error de compilcación:
The field Examen.calificacion is not visible
De igual modo el encapsulamiento nos permite ocultar métodos que son utilizados internamente por nuestra clase para no exponerlos a otras clases, un ejemplo de esto son los métodos:

```
private float sumarAciertos()
private float generarPromedio()
```

Sobreescritura

La sobreescritura de métodos permite cambiar el comportamiento de un método heredado de una clase padre.

Como se puede ver la clase **Figura** define el método **calcularArea**() , pero una figura por si misma no puede calcular un área, quien sabe como hacerlo son las clases **Circulo** y **Cuadrado**, por esto sobreescriben el método, a continuación algunas reglas:

-Los **argumentos** del método deben ser **exactamente los mismos** del que va a sobreescribir
-El **valor de retorno** debe ser el **mismo o un subtipo** del que se desea sobreescribir
-El **nivel de acceso** no puede ser más **restrictivo** que el que se va a sobreescribir
-Los métodos **final** no se pueden sobreescribir

Sobrecarga

La sobrecarga de métodos permite tener 2 métodos con el mismo nombre, la única regla es que deben recibir parámetros diferentes, un ejemplo de esto son los métodos :

```
public float presentarExamen(ExamenNormal e )

public float presentarExamen(ExamenExtraordinario e )
```

Los dos métodos se encuentran en la clase Estudiante y ambos reciben parámetros distintos.

La sobrecarga se puede aplicar también a **constructores**, permitiendo crear nuestros objetos de diferentes formas de acuerdo al problema, a continuación se presentan algunas reglas:

-Los métodos deben tener el **mismo nombre**
-Los métodos deben recibir **diferentes parámetros**
-Los métodos pueden tener **diferentes** valores de **retorno**
-Los métodos pueden arrojar **diferentes excepciones**

< DEVS4J >
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j

Constructores, static, interfaces, etc.

< DEVS4J >
Hecho por Devs para Devs

Constructores

Los objetos se crean a través de constructores, estos tienen las siguientes características:

-No tienen valor de retorno
-Su nombre es el mismo de la clase
-Es posible tener multiples constructores siempre y cuando estén sobrecargados.
-Pueden tener cualquier modificador de acceso.
-Si no se declara un constructor Java creará uno por defecto
-El constructor por defecto no recibe parámetros
-Si se declara un constructor, se renunciará al constructor por defecto

Ejemplo:

```
public class Perro {
	private String nombre;
	private String raza;

	public Perro() {
	}

	public Perro(String n, String r) {
		nombre = n;
		raza = r;
	}
	.....
}
```

La clase anterior tiene 2 constructores, uno que no recibe parámetros y el otro que recibe como parámetro el nombre y la raza del perro. Las siguientes son formas válidas para crear un objeto de tipo Perro:

```
Perro p=new Perro();
Perro p2=new Perro("Roscko","Terrier escocés");
```

Invocación de un constructor desde otro

Es posible invocar un constructor desde otro, para hacerlo, se hace uso de la palabra reservada **this**, veamos la clase **Perro** modificada:

```
public class Perro {
	private String nombre;
	private String raza;

	public Perro() {
		this("NombrePorDefecto","RazaPorDefecto");
	}

	public Perro(String n, String r) {
		nombre = n;
		raza = r;
	}
	.....
}
```

En el ejemplo anterior el constructor que no recibe parámetros invoca al que si recibe y asígna un nombre y una raza por defecto, cuando se hace uso de la palabra reservada **this** para invocar a otro constructor, esta debe ser la primera línea de código.

Invocación a un constructor de la clase padre

Todos los constructores ejecutan al constructor de su clase padre antes de ejecutarse, incluso si nosotros no lo definimos en el código tendremos una llamada como la siguiente:

```
public Perro() {
 super();
}
```

super() al igual que **this**() se utiliza para ejecutar un constructor, la diferencia entro uno y otro es que this ejecuta uno en la clase actual y super uno en la clase padre.

Static

La palabra reservada **static** se utiliza para crear variables y métodos que existirán independientemente de si existe o no una instancia(objeto) de una clase. Existirá solo una copia de un método / atributo static sin importar el número de objetos que tenga la clase.

Variables static

Cuando se tiene una variable **static** se tendrá solo una copia de esta para todas las instancias, veamos el siguiente ejemplo:

```
public class Humano {
	static int numeroDeHumanos = 0;
	public Humano() {
		numeroDeHumanos++;
	}
	public static void main(String[] args) {
		new Humano();
		new Humano();
		new Humano();
		System.out.println(numeroDeHumanos);
	}
}
```

Salida : 3

Del código anterior podemos ver lo siguiente:

-La variable numeroDeHumanos existe sin importar si hay objetos o no creados, inicialmente tiene un valor de cero.
-Cada que se crea un objeto Humano se aumenta en uno
-Es posible acceder a ella sin necesidad de crear un objeto
-Su valor es compartido por todos los objetos
-Puedes acceder a una variable static que se encuentra en otra clase del siguiente modo **NombreClase**.variable.

Uso de la palabra reservada this

La palabra reservada **this** se utiliza para referenciar al objeto actual, veamos el siguiente ejemplo :

```
public class Perro {
	private String nombre;
	private String raza;

	public Perro(String nombre, String raza) {
		nombre = nombre;
		raza = raza;
	}
	.....
}
```

En este ejemplo las asignaciones no funcionan dado que los parámetros se asignan a si mismos y no a los atributos de la clase, para indicar que se desea asignar a los atributos de la clase utilizaremos la palabra reservada this del siguiente modo:

```
public class Perro {
	private String nombre;
	private String raza;

	public Perro(String nombre, String raza) {
		this.nombre = nombre;
		this.raza = raza;
	}
 .....
}
```

Cohesión

El término cohesión es utilizado para indicar el grado en el que una clase tiene **un solo proposito** bien definido.

Una clase bien enfocada tendrá **alta cohesión**, mientras más cohesión tengamos nuestras clases serán más fáciles de mantener.

Métodos static

Es posible ejecutar un método static sin necesidad de crear un objeto, veamos el siguiente ejemplo:

```
public class Calculadora {

	static int suma(int x, int y) {
		return x + y;
	}

	static int resta(int x, int y) {
		return x - y;
	}

	public static void main(String[] args) {
		System.out.println(suma(10, 20));
	}
}
```

Podemos ver que desde el método main se invoca al método suma sin necesidad de crear un objeto de la clase Calculadora,

Interfaces

Cuando implementas una interfaz aceptas el contrato de implementar sus métodos.

Una interfaz define métodos sin implementación , constantes y métodos default, veamos el siguiente ejemplo:

```
public interface Dibujable {
	void dibujar();
}
```

Las interfaces deben ser adjetivos, veamos algunas implementaciones de la interfaz Dibujable:

```
public class Rectangulo implements Dibujable{
	@Override
	public void dibujar() {
		System.out.println("Dibujando rectangulo");
	}
}
public class Pared implements Dibujable {
	@Override
	public void dibujar() {
		System.out.println("Dibujando en la pared");
	}
}
```

Las clases Rectangulo y Pared implementan la interfaz dibujable, por esto deben sobre escribir el método dibujar.

A partir de la versión Java 8 es posible utilizar default methods, los cuales permiten definir una implementación en interfaces, veamos el siguiente ejemplo:

```
interface A{
	default void foo(){
		System.out.println("Método default");
	}

	void bar();
}
```

En el ejemplo anterior tenemos 2 métodos, foo() es un método default, por tanto la clase que implemente la interfaz A no esta obligada a implementarlo, mientras que bar() debe ser implementado.

Acoplamiento

Acoplamiento es el grado en el que una clase conoce sobre otra. Una clase A que depende de una clase B para funcionar tiene un alto acoplamiento.

El alto acoplamiento complica el mantenimiento de las aplicaciones, por esto siempre buscaremos reducirlo

Arreglos, clases abstractas y enumeraciones

DEVS4J
Hecho por Devs para Devs

Clases abstractas

La definición de un método sin implementación se conoce como método **abstracto**. Una clase abstracta es una clase que puede contener métodos abstractos y métodos concretos:

```
public abstract class Figura {
	private String material;

	public String getMaterial() {
		return material;
	}
	public void setMaterial(String material) {
		this.material = material;
	}
	public abstract double calcularArea();
}
public class Circulo extends Figura {
	private double radio;
	@Override
	 public double calcularArea() {
		return Math.PI * Math.pow(radio, 2);
	 }
	 //Getters y setters de radio
}
public class Cuadro extends Figura {
	private double lado;
	@Override
	public double calcularArea() {
		return lado*lado;
	}
	 //Getters y setters de lado
}
```

Dado que la figura por si sola no sabe como realizar un cálculo de área el método **calcularArea** es definido como abstracto, la implementación debe implementar ese método.

Reglas importantes:

- -Una clase abstracta no puede ser **final**.
- -Los métodos abstractos deben ser **implementados** por las **subclases**
- -**No** se pueden crear **objetos** de una clase abstracta
- -Es posible definir **constructores** y métodos static en la clase abstracta.
- -Puede tener **métodos final** para evitar que las implementaciones los cambien.
- -Un método **abstracto no** puede ser definido como **final**.
- -Si una clase abstracta implementa una **interfaz**, puede **no implementar** sus métodos, pero la clase que herede de ella debe implementarlos.
- -Los métodos abstractos **no** pueden ser **static**.
- -Una interfaz es una clase 100% abstracta.

Identificadores

Un **identificador** es el nombre que asignamos a las cosas en Java, estos deben seguir las siguientes reglas:

- -Pueden **iniciar** con una **letra**, **guión bajo** o **símbolo de moneda**.
- -Después del primer caracter se pueden **incluir números**.
- -Pueden tener **cualquier longitud**.
- -**No** se pueden utilizar **palabras reservadas** como identificadores.
- -Los identificadores son case sensitive, FOO y foo son 2 identificadores diferentes.

Java beans standards

-Si se desea obtener un valor el método debe tener el prefijo get, por ejemplo **getName**();

-Si se desea obtener un valor boolean también es válido utilizar el prefijo is, por ejemplo **isAlive**();

-Si se desea asignar un valor se debe utilizar el prefijo set, por ejemplo **setName**(String name);

-Los métodos getter / setter deben ser públicos

Arreglos

En Java los arreglos son objetos que almacenan **multiples** valores del **mismo tipo**, una vez definido el tamaño este no puede cambiar, veamos algunos puntos a considerar:

- -Pueden almacenar primitivos y objetos.
- -Existen arreglos unidimensionales y multidimensionales

Iterando sobre un arreglo

Es posible iterar a cada elemento de un arreglo con el siguiente código:

```
int numeros[] = { 10, 11, 2, 12, 3, 34, 2, 12 };
for (int i = 0; i < numeros.length; i++) {
	System.out.println(numeros[i]);
}
```

Es posible hacerlo de la misma forma con un ciclo while y do while pero existe un tipo de cilo for especial llamado **forEach**, a continuación un ejemplo:

```
int numeros[] = { 10, 11, 2, 12, 3, 34, 2, 12 };
for (int numero : numeros) {
	System.out.println(numero);
}
```

Este tipo de ciclo funciona a través de una **variable temporal** que va tomando el valor de cada elemento del arreglo, **forEach** es una opción cuando:

- -No necesitamos modificar el arreglo
- -No necesitamos indices

Arreglos de objetos

Es posible tener tanto arreglos de tipos primitivos como de objetos, a continuación un ejemplo:

```
public class Taco {
	private String sabor;
	private float precio;
	private boolean cebolla;
	private boolean cilantro;

	public Taco(String sabor, float precio, boolean cebolla, boolean cilantro) {
		this.sabor = sabor;
		this.precio = precio;
		this.cebolla = cebolla;
		this.cilantro = cilantro;
	}
	 //Getters y setters
}
public class Taqueria {
	public static void main(String[] args) {
		Taco orden1[] = new Taco[3];
		orden1[0] = new Taco("Pastor", 10.5f, true, true);
		orden1[1] = new Taco("Suadero", 20.5f, false, true);
		orden1[2] = new Taco("Bistec", 30.0f, true, true);
		float precioTotal = 0.0f;
		for (Taco taco : orden1) {
			precioTotal += taco.getPrecio();
		}
		System.out.println("El precio de la orden es de : " + precioTotal);
	}
}
```

Arreglos multidimensionales

Los arreglos pueden tener multiples dimensiones a continuación un ejemplo:

```
public static void main(String[] args) {
	int arrays[][] = { { 2, 2 }, { 4, 4 }, { 5, 5 } };
	for (int[] array : arrays) {
		for (int value : array) {
			System.out.print("\t " + value);
		}
		System.out.println();
	}
}
```

Declaración de arreglos

Es posible declarar una referncia a un array de cualquiera de las siguientes formas:

Arreglos de primitivos

- int arr [];
- int [] arr;

Arreglos de objetos

- String arr[];
- String []arr;

Inicialización de arreglos

Existen diferentes formas de crear los arreglos, veamos las siguientes:

-Forma implícita

```
String nombres[] = { "Alex", "Pedro", "Juan" }
```

-Forma explícita

```
String nombres[] =new String [3];
nombres[0]="Alex";
nombres[1]="Pedro";
nombres[2]="Juan";
```

-Forma anónima

```
String nombres[] = new String[] { "Alex", "Pedro", "Juan" };
```

Acceso a arreglos

Para acceder a los elementos de un arraglo lo haremos a través de su indice, veamos el siguiente ejemplo:

```
	String nombres[] = { "Alex", "Pedro", "Juan" }
	System.out.println(nombres[0]);
```

Imprimirá : Alex

Enumeraciones

Una enumeración es un tipo especial de clase que representa un grupo de constantes, estas constantes deberán estar en mayúsculas:

```
public enum SaborPalomitas {
	CHILE, MANTEQUILLA, QUESO, CARAMELO;
}
```

Para acceder al valor de la enumeración se realizará del siguiente modo:

```
SaborPalomitas.CARAMELO
```

Es posible asignarle valores a las enumeraciones, veamos el siguiente

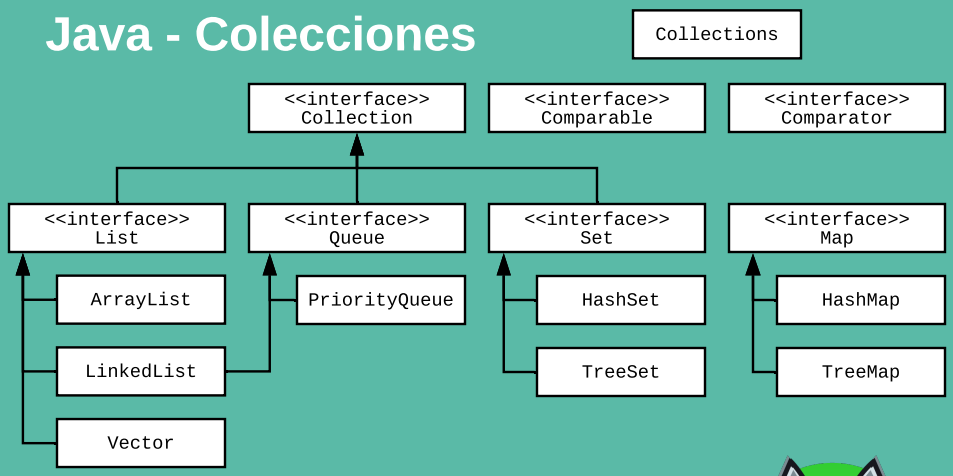
```
public enum TamanioDePalomitas {
	CHICA(20.5f), MEDIANA(30.1f), GRANDE(50.0f);
	private float precio;
	private TamanioDePalomitas(float precio) {
		this.precio = precio;
	}
	public float getPrecio() {
		return precio;
	}
}
```

Para acceder a un valor de la enumeración se realizará del siguiente modo:

```
TamanioDePalomitas tamanioMediano = TamanioDePalomitas.MEDIANA;
System.out.println(tamanioMediano.getPrecio());
```

Al ejecutar lo anterior tendremos la siguiente salida: **30.1**

Java - Colecciones



Wrapper classes

Las **wrapper clases** en Java tienen 2 principales propósitos:

-Proveer un mecanísmo para envolver valores primitivos en objetos y así ser incluidos en actividades exclusivas para objetos.

-Proveer una gran variedad de utilidades, muchas de ellas están relacionadas con la conversión.

Equivalencias de wrapper classes con primitivos

Existe una **wrapper class** para cada uno de los tipos primitivos:

-boolean- Boolean
-byte - Byte
-char - Character
-double - Double
-float - Float
-int - Integer
-long - Long
-short - Short

Todas las **wrapper classes** excepto **Character** proveen 2 constructores, uno que toma el valor primitivo y otro que toma la representación en **String**.

Métodos de conversión

Existen diferentes métodos que nos permiten realizar actividades de conversión, a continuación se muestran algunos de los más utilizados:

-**xxxValue**() : Se utiliza cuando necesitas obtener el valor primitivo para un valor dado:

```
Integer val=new Integer(11);
double doubleValue = val.doubleValue();
System.out.println(doubleValue);
```

-**parseXXX**() : Convierte un String a un tipo primitivo:

```
int intValue = Integer.parseInt("11");
System.out.println(intValue);
```

-**valueOf**() : Recibe un String y devuelve un Wrapper del tipo invocado:

```
Double doubleValue2 = Double.valueOf("3.14");
System.out.println(doubleValue2);
```

Autoboxing y Auto-unboxing

A partir de la versión 5 de Java existen las siguientes características:

-**Autoboxing** : Es la conversión que hace el compilador de **Java** de el tipo primitivo o a su correspondiente **wrapper class**:

```
Integer x=10;
```

-**Auto-unboxing** : Es la conversión que hace el compilador de **Java** de la **wrapper class** al tipo de dato primitivo:

```
Integer x=10;
int y=x;
```

== , equals()

```
if (i1 != i2) {
 System.out.println("Objetos diferentes");
}
if (i1.equals(i2)) {
 System.out.println("Objetos equivalentes");
}
```

Produce la salida:

Objetos diferentes
Objetos equivalentes

Recordemos que == validan si ambas referencias apuntan al mismo objeto, mientras que equals verifica si los objetos son equivalentes.

Colecciones en Java

Las colecciones en **Java** son un **framework** que permite almacenar un grupo de objetos, a diferencia de los arreglos las colecciones pueden crecer, veamos un resumen de las interfaces principales:

-**Collection** : Todas las colecciones a excepción de las que provienen de la interfaz **Map** provinen de la interfaz Collection.

-**List** : Permite generar listas simples de objetos

-**Set** : Permite almacenar objetos únicos

-**Map** : Permiten almacenar objetos en pares llave y valor, las llaves deben de ser únicas

-**Queue** : Permiten almacenar objetos de acuerdo al orden de inserción o a reglas definidas

ArrayList

Colección basada en arreglos que puede crecer, cuando es instanciada tiene una capacidad default de 10 elementos, conforme se van agregando valores la capacidad se va incrementando.

ArrayList tiene una velocidad constante al agregar y al obtener elementos.

ArrayList no es muy eficiente cuando se tiene que agregar en una posición específica o remover un elemento.

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

LinkedList

LinkedList es una implementación de una lista doblemente ligada, su performance es mejor a un ArrayList cuando se tiene que agregar o remover un elemento, pero es peor cuando se quiere obtener o modificar un elemento.

LinkedList implementa también la interfaz Queue así que se puede utilizar al mismo tiempo como una estructura de tipo FIFO.

```
LinkedList<String> lista = new LinkedList<>();
```

Vector

Vector junto con **HashTable** son las colecciones iniciales agregadas en la versión 1.2, su comportamiento es similar al de un **ArrayList** a diferencia que **Vector** es thread safe.

```
Vector<String> lista = new Vector<>();
```

PriorityQueue

Esta clase fue agregada en **Java** 5, como **LinkedList** soporta el comportamiento default de una estructura **FIFO**, **PriorityQueue** ordena los elementos de acuerdo a una prioridad, de este modo podemos acceder primero a los elementos que tienen una mayor prioridad.

```
PriorityQueue<String> queue = new PriorityQueue<>();
```

DEVS4J
Hecho por Devs para Devs

HashSet

Colección utilizada cuando necesitas un conjunto de datos sin elementos duplicados y no te importe el orden de los datos.

```
HashSet<String> set = new HashSet<>();
```

TreeSet

TreeSet es una de las colecciones ordenadas e implementa una estructura de tipo árbol rojo-negro y garantiza que los elementos serán únicos y estarán ordenados de forma ascendente de acuerdo a un orden natural.

```
TreeSet<String> set = new TreeSet<>();
```

HashMap

Utiliza **HashMap** cuando requieras una colección que contenga pares, llave-valor, las llaves pueden contener null pero solo un elemento. **HashMap** no permite llaves duplicadas.

El acceso a los datos se basa en el hashCode, entre mejor sea la implementación de hashCode, mejor será el performance.

```
HashMap<Integer,String> map = new HashMap<>();
```

TreeMap

TreeMap es una de las colecciones ordenadas al igual que **TreeSet**, solo que el orden lo definirán las llaves y no los valores.

```
TreeMap<Integer,String> map = new TreeMap<>();
```

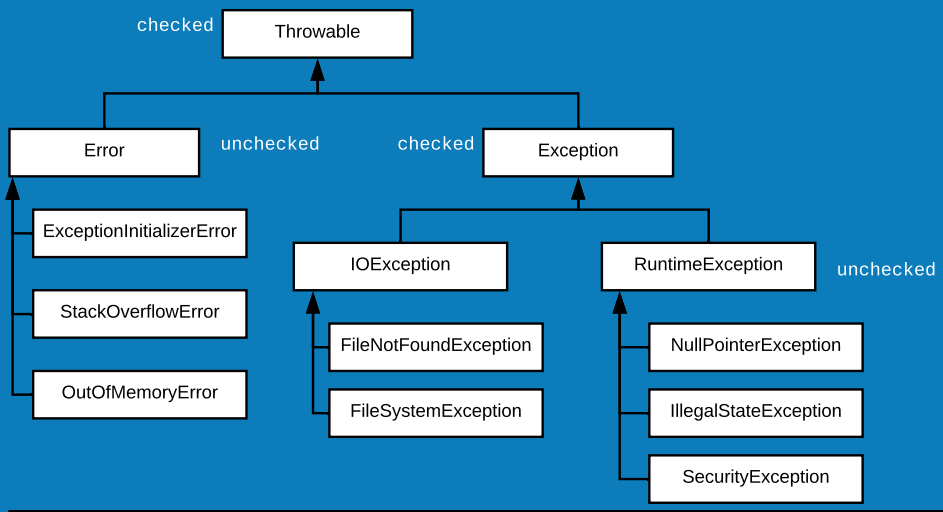
Comparable / Comparator

Se utiliza **Comparable** y **comparator** para definir el orden en colecciones como TreeSet y TreeMap.

Collections

La clase java.util.**Collections** contiene un conjunto de métodos **static** útiles para trabajar con colecciones.

Java - Manejo de Errores



El diagrama anterior muestra algunas de las clases más importantes al momento de manejar errores:

-**Throwable** : Es la clase padre de todos los problemas que encontraremos al desarrollar aplicaciones Java.

-**Error** : Se utiliza para representar situaciones inusuales de las que las aplicaciones no se pueden recuperar.

-**Exception** : Son errores en tiempo de ejecución de nuestros programas y se pueden clasificar de acuerdo a lo siguiente:
  -**Checked** : Son excepciones que el compilador requiere que manejemos, las utilizamos cuando quien llama a nuestro método se puede recuperar de esa situación.
  -**Unchecked** : Son excepciones que el compilador no requiere que manejemos, todas las excepciones que hereden de **RuntimeException** serán consideradas como unchecked. Las utilizaremos para representar problemas que resultan de errores de programación.

try - catch -finally

A continuación se muestra un ejemplo sobre el uso de **try-catch-finally**:

```
try {
 /**
  * Código que puede generar excepciones
  */
} catch (Exception e) {
 /**
  * Código a ejecutar en caso de que
se genere una excepción
  */
} finally {
 /**
  * Código a ejecutar sin importar si se
  * generó o no una excepción
  */
}
```

Union catch block

Permite tener un solo catch para múltiples excepciones:

```
try {
/**
 * Código que puede generar una excepción
 */
} catch (FileNotFoundException |
FileAlreadyExistsException e) {
/**
 * Código que se ejecuta en caso de que se genere una
 * FileNotFoundException o FileAlreadyExistsException
 */
}catch (IOException e){
/**
 * Código que se ejecuta en caso de que se genere una
 * IOException
 */
}
```

try with resources

Try with resources se incluyó en Java 7 y nos permiten declarar recursos que se utilizarán en un bloque try con la seguridad de que serán cerrados después de la ejecución del bloque, Los recursos declarados deben implmentar la interfaz **AutoClosable**.

Reglas al utilizar try-catch-finally

A continuación se muestran algunas reglas al utilizar **try-catch-finally**:

-Solo el bloque **try** es obligatorio.
-Es posible definir un bloque con solo un **try** y un **catch** del siguiente modo:

```
try{

}catch(Exception e){

}
```

-Es posible definir un bloque con solo un **try** y un **finally** del siguiente modo:

```
try{

}finally{

}
```

-Un **try** puede tener muchos catch pero solo uno se ejecutará.
-Se deben colocar los **catch** de la excepción más específica a la más general.
-El bloque **finally** se ejecutará incluso si en el **catch** se tiene una sentencia return.
-La única forma de evitar que se ejecute un bloque **finally** es declarando un **System**.exit(..);

Uso de try-with-resources

La clase **PrintWriter** se utiliza para escribir en un archivo, veamos el siguiente ejemplo:

```
try (PrintWriter writer = new PrintWriter(new
File("test.txt"))) {

    writer.println("Hello World");

} catch (FileNotFoundException e) {
    //Manejo de la excepción
}
```

En el código anterior sin importar si existe o no una excepción al escribir en el archivo se ejecutará el método close definido en la interfaz **Autoclosable** e implementado por **PrintWriter**.

Creación de tu propio Autoclosable

Para construir tu propio recurso que puede ser manejado por **try-with-resources** debes crear una clase que implemente **Closable** o **Autoclosable** y sobre escribir su método close como se muestra:

```
public class MiRecurso implements AutoCloseable {
    public void imprimirMensaje() {
        System.out.println("Esta es una acción");
    }
    @Override
    public void close() throws Exception {
        System.out.println("Se esta cerrando mi recurso");
    }
}
Una vez hecho esto podrás utilizarlo del siguiente modo:
try(MiRecurso recurso=new MiRecurso()){
    recurso.imprimirMensaje();
} catch (Exception e) {
    //Manejo de la excepción
}
```

Creación de tu propia Excepción

Si las excepciones que proporciona Java no representan el error que necesitas, es posible crear tu propia Excepción, para esto deberás de crear una clase que herede de **Exception** o de **RuntimeException** dependiendo si quieres que tu excepción sea checked o unchecked:

```
class MiExcepcionNoChecada extends RuntimeException {
}
class MiExcepcionChecada extends Exception {
}
```

Lanzamiento de excepciones

Si existe una parte en tu código donde deseas lanzar una excepción se utilizará la palabra reservada **trow** como se muestra a continuación:

```
class MenorDeEdadException extends Exception{
}
public void validarEdad(int edad) throws
MenorDeEdadException {
    if (edad < 18) {
        throw new MenorDeEdadException();
    }
}
```

Uso de la palabra reservada throws

Se utiliza la palabra reservada throws para indicar que en caso de que se genere una excepción esta no será manejada por el método actual sino por el que lo mando llamar:

```
public void readFile() throws FileNotFoundException {
    File file = new File("archivo_que_no_existe.txt");
    FileInputStream stream = new FileInputStream(file);
 /**
  * código que utiliza stream
  */
}
```

Como se puede ver, a pesar de que **FileNotFoundException** es una excepción de tipo checked, no es necesario colocar el código dentro de un **try catch** ya que quien será responsable de cachar la excepción es el método que invoque al método **readFile**().

Utilizaremos este método para delegar el manejo del error a quien tiene la capacidad de manejarlo, por ejemplo si hay un componente que solo accede a datos puede propagar el error hasta el componente que maneja la presentación para que le muestre un mensaje adecuado al usuario.

Reglas al utilizar throws / throw

-**Throws** es necesario solo para excepciones de tipo **checked**
-Cuando se sobrescribe un método no es posible agregar throws con una excepción nueva de tipo **checked**
-Cuando se llega a una línea con **throw** la ejecución se detiene y se comienza a propagar la excepción
-Si la excepción no es manejada por ningún método, será manejada por la **JVM**
-Un método puede declarar más de una excepción con la palabra reservada **throws**
-Se puede utilizar **throw** solo con objetos de tipo **Throwable**

Java - clases internas

Inner classes

Las clases internas se pueden clasificar del siguiente modo:

-Clases internas regulares
-Clases internas a nivel de método
-Clases anónimas
-Clases internas estáticas

Clases internas regulares

Las clases internas representan clases que no son anónimas, static o a nivel de método. A continuación se presenta un ejemplo:

```
public class External {
	void foo() {
		System.out.println("Foo");
	}
	class Internal {
		void bar() {
			System.out.println("Bar");
		}
	}
}
```

Es posible crear instancias de clases internas del siguiente modo:

```
External external = new External();
External.Internal internal = external.new Internal();
external.foo();
internal.bar();
```

Clases internas estáticas

A continuación se presenta un ejemplo de una clase estática interna:

```
public class StaticExternal {
	static class Interna {
		void foo() {
			System.out.println("Foo");
		}
	}
	public static void main(String[] args) {
		Interna internal = new Interna();
		internal.foo();
	}
}
```

No es necesario crear objetos de la clase externa para instanciar objetos de la clase interna.

Clases a nivel de método

También es posible declarar clases a nivel de método:

```
public class MethodLevelClasses {
	static void print() {
		class InnerMethodClass {
			void foo() {
				System.out.println("test");
			}
		}
		InnerMethodClass innerClass = new
		InnerMethodClass();
		innerClass.foo();
	}
public static void main(String[] args) {
	print();
}
```

Clases anónimas

Son clases sin nombre que heredan de la clase especificada:

```
abstract class Test {
	abstract void foo();
}
public class AnonymousClass {
	public static void main(String[] args) {
		Test object = new Test() {
			@Override
			void foo() {
			System.out.println("Foo");
			}
		};
		object.foo();
	}
}
```

Clases anónimas con interfaces

Son clases sin nombre que también pueden implementar una interfaz

```
interface Foo {
	void foo();
}
public class InterfaceExample {
	public static void main(String[] args) {
		Foo f = new Foo() {
			@Override
			public void foo() {
				System.out.println("Foo");
			}
		};
		f.foo();
	}
}
```

Threads

Un hilo es un objeto en java que tiene variables y métodos. Lo especial de esta clase es que permite ejecutar tareas de forma concurrente.

Existen 2 formas de definir un hilo:

-Creando un objeto de la clase **Thread** y sobreescribiendo el método **run**()

-Creando un objeto que implemente la interfaz **Runnable** e implementando el método **run**()

Definición de Threads a través de herencia

A continuación se muestra un ejemplo de la definición de un hilo utilizando herencia:

```
class DescendingCounter extends Thread {
	private int maxNumber;
	public DescendingCounter(int maxNumber) {
		this.maxNumber = maxNumber;
	}

	@Override
	public void run() {
		for (int i = maxNumber; i >= 0; i--) {
			try {
				Thread.sleep(1000);
				System.out.println("Desc:" + i);
			} catch (InterruptedException e) {
				System.err.println(e);
			}
		}
	}
}
```

El código anterior muestra como hacer un contador de forma descendente.

Definición de Threads a través de implementación

A continuación se muestra un ejemplo de la definición de un hilo implementando la interfaz **Runnable**:

```
class AscendingCounter implements Runnable {

	private int maxNumber;

	public AscendingCounter(int maxNumber) {
		this.maxNumber = maxNumber;
	}

	@Override
	public void run() {
		for (int i = maxNumber; i >= 0; i--) {
			try {
				Thread.sleep(800);
				System.out.println("Asc : " + i);
			} catch (InterruptedException e) {
				System.err.println(e);
			}
		}
	}
}
```

El código anterior muestra como hacer un contador ascendente.

Ejecución de un Thread

El siguiente ejemplo muestra como ejecutar hilos:

```
public class Counters {
	public static void main(String[] args) {
		DescendingCounter descending = new
		DescendingCounter(10);

		Thread ascending = new Thread(new
		AscendingCounter(10));
		descending.start();
		ascending.start();
lkvc mclkvm
	}
}
```

Del código anterior podemos ver los siguientes puntos:

-Para ejecutar un hilo **NO** ejecutaremos su método **run**().

-Para ejecutar un hilo ejecutaremos su método **start**().

-Si ejecutamos el método **run**() ejecutará el código, pero no de forma concurrente, sino como un método común.

-El método void sleep(long millis) permite dormir el hilo la cantidad de milisegundos especificada.

-Si se crea un hilo a través de una clase que hereda de **Thread** solo se debe ejecutar su método **start**().

-Si se crea un hilo a través de una clase que implementa **Runnable** el objeto se debe pasar como parámetro a un objeto de la clase **Thread**.

-***"End of creation"*** se imprimirá antes de que se terminen de ejecutar los dos procesos concurrentes.

Ciclo de vida de los hilos

Los hilos se pueden encontrar en los siguientes estados:

-**New**: El hilo se ha creado pero no se ha ejecutado el método start()

-**Runnable**: Se ejecutó el método start() y el hilo está listo para ser ejecutado.

-**Running**: El hilo se encuentra en ejecución.

-**Waiting/blocking**:El hilo no es elegible para ser ejecutado. El hilo esta vivo pero no es elegible.

-**Dead**: El hilo se considera muerto cuando termina la ejecución del método run. No es posible iniciar de nuevo un hilo que se encuentra en estado dead.

Thread scheduler

Es la parte de la maquina virtual de Java que decide los hilos que se van a ejecutar de acuerdo a sus prioridades.

< DEVS4J >
Hecho por Devs para Devs

Hilos parte 2, Lambdas, Optional, Streams y Sockets

< DEVS4J >
Hecho por Devs para Devs

Ciclo de vida de los hilos

Los hilos se pueden encontrar en los siguientes estados:

-**New**: El hilo se ha creado pero no se ha ejecutado el método **start**().

-**Runnable**: Se ejecutó el método **start**() y el hilo está listo para ser ejecutado.

-**Running**: El hilo se encuentra en ejecución.

-**Waiting/blocking**:El hilo no es elegible para ser ejecutado. El hilo esta vivo pero no es elegible.

-**Dead**: El hilo se considera muerto cuando termina la ejecución del método **run**(). Un hilo en estado ***dead*** no se puede volver a iniciar.

Métodos últiles

-**start**() :Inicia la ejecución de un hilo

-**join**() : Hace que un hilo espere la ejecución de otro hasta que esta termine.

-**sleep**(long milis) : Método estático que puedes utilizar para alentar la ejecución de un hilo.

-**yield**(): Puede causar que se mande al hilo a estado ***runnable***, dependiendo de su prioridad.

-**wait**() : Envía al hilo a esperar en estado ***waiting***.

-**notify**() : Notifica para que un hilo en estado ***waiting*** pase a estado ***runnable***.

-**notifyAll**() : Notifica a todos los hilos en estado ***waiting*** para que vuelvan a estado ***runnable***.

-**setName**(String name) : Asigna un nombre al hilo.

-**setPriority**(int priority) : Define la prioridad de un hilo

Lambdas

Las expresiones **lambda** son una de las adiciones más importantes que se hicieron a la versión 8 de Java y proveen una forma simple de representar un método a través de una expresión.

Las expresiones lambda son implementaciones de **interfaces funcionales**. Una interfaz funcional es básicamente una interfaz con un solo método abstracto.

Sintaxis

Una expresión lambda está compuesta por las siguientes 3 partes:

lista de argumentos : (int x, int y)

Token : ->

Cuerpo : x + y

Creación de un Comparator a través de lambdas

A continuación se muestra un ejemplo de un **Comparator** haciendo uso de lambdas:

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona("Juan", "López"));
personas.add(new Persona("Arturo", "Sánchez"));

Comparator<Persona> comparator = (Persona p, Persona p2) ->
p.getNombre().compareTo(p2.getNombre());

Collections.sort(personas, comparator);
```

Creación de hilos con lambdas

Un ejemplo de una interfaz funcional es la interfaz **Runnable** la cuál solo define el método **run**(). A continuación un ejemplo:

```
Runnable counter = () -> {
    for (int i = 0; i <= 100; i++) {
        try {
            Thread.sleep(800);
            System.out.println("Asc : " + i);
        } catch (InterruptedException e) {
            System.err.println(e);
        }
    }
};
```

El código anterior representa la creación de un objeto que implementa la interfaz **Runnable**.

Implementación de una interfaz propias

A continuación se muestra un ejemplo de la implementación de una interfaz haciendo uso de lambdas:

```
interface Calculable {
    double avg(double... numbers);
}

public class InterfaceImplementation {
    public static void main(String[] args) {
        Calculable calc = (numbers) -> {
            double sum = 0.0;
            for (int i = 0; i < numbers.length; i++) {
                sum += numbers[i];
            }
            return sum / numbers.length;
        };
        System.out.println(calc.avg(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
    }
}
```

Clase Optional

Esta clase nos permite representar valores opcionales en lugar de valores nulos. Veamos el siguiente ejemplo:

```
public Integer findValue(Integer value, Integer[] array) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == value) {
            return i;
        }
    }
    return null;
}
```

El método anterior busca un valor en un arreglo y devuelve su posición, en caso de que no se encuentre devuelve **null**. A continuación se presenta un ejemplo utilizando **Optional**:

```
public static Optional<Integer> findValue(Integer value, Integer[] array) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == value) {
            return Optional.of(i);
        }
    }
    return Optional.empty();
}
```

Esto hace una solución más limpia y menos propensa a bugs. El siguiente ejemplo muestra como manejar una respuesta opcional:

```
Optional<Integer> result =  findValue(5, new Integer[] { 10, 33, 23, 57, 88 });
if(result.isPresent()){
    System.out.println(result.get());
}else{
    System.out.println("No se encontro el resultado");
}
```

Java 8 Streams

Un ***stream*** es una pieza de código que se enfoca en conjuntos y no en piezas individuales. Supongamos el siguiente problema:

Se desean obtener las 3 primeras personas mayores de 18 años que se encuentran en una lista:

```
int count = 0;
List<Persona> resultado = new ArrayList<>();
for (Persona persona : personas) {
    if (persona.getEdad() >= 18) {
        resultado.add(persona);
        count++;
        if(count==3){
            break;
        }
    }
}
```

En el código anterior se da una solución al problema de forma imperativa. A continuación se presenta una solución de forma funcional a través de **streams**:

```
List<Persona> primerosMayores =
personas.stream().filter(p -> p.getEdad() >= 18).limit(3)
  .collect(Collectors.toList());
```

¿Cómo utilizar streams?

Para utilizar streams seguiremos los siguientes pasos:

1: Iniciar con una implementación concreta : **Arrays**, **Set**, **List**, **Map**, etc.

2: Ejecutar el método **stream**(), es posible concatenar múltiples streams.

3: Utilizar operaciones entre streams entre las que tenemos : **filter**, **map**, **reduce**, etc.

4: Volver al caso concreto, este puede ser : **Integer**, **List**, **Map**, **Set**, **Optional**, etc.

5: Para volver al caso concreto podremos utilizar algunas de las siguientes operaciones: **sum**(), **collect**(Collectors.**toList**()), **average**(), **collect**(Collectors.**groupBy**()), etc.

Sockets

Los **sockets** permiten escribir programas que se ejecutan en diferentes computadoras que estan conectadas a través de la red.

Un socket esta conformado por:

-**Un protocolo** : Puede ser UDP o TCP

-**Una IP** : El lugar de origen o destino al que se enviará la información

-**Un puerto**: Son utilizados para determinar a donde dirigir el tráfico del 0 al 1023 son puertos dedicados del sistema.

ServerSocket

Nos permite definir a nuestro **socket** servidor. A continuación se presenta un ejemplo:

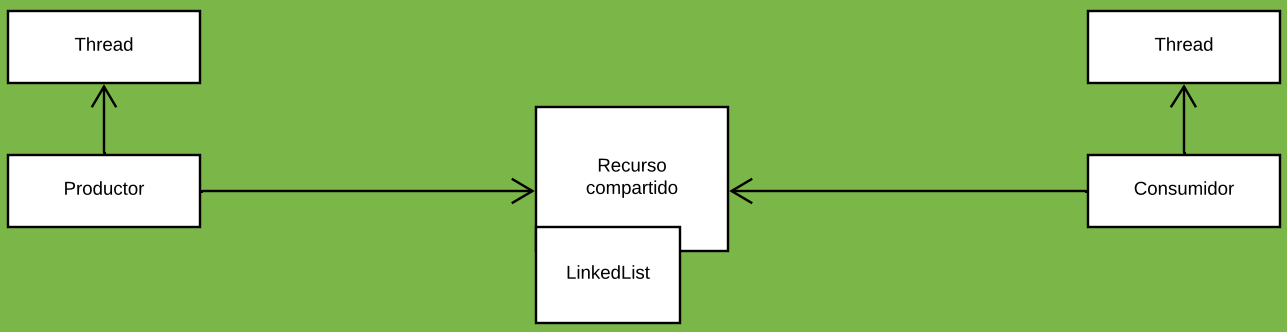
```
ServerSocket server = new ServerSocket(8080);
```

Socket

Nos permite definir a los clientes que se conectaran al servidor. A continuación se presenta un ejemplo:

```
Socket socket = new Socket("localhost", 8080);
```

Productor consumidor



Notas

- Los productores deben producir mensajes en el **LinkedList.**

-El objeto que contiene al **LinkedList** será un recurso compartido entre el producer y el consumer.

-Si no hay mensajes a consumir el consumidor debe esperar hasta que sea notificado que hay un mensaje nuevo.

-Si el productor genera un mensaje debe notificar a los consumidores para que vayan a consumirlo.

Todo esto se debe hacer de forma sincronizada para que los hilos se ejecuten de forma infinita.

Pueden existir N productores y N consumidores.

Uso de streams

DEVS4J
Hecho por Devs para Devs

XXStream (range y rangeClosed)

Puedes utilizar IntStream, DoubleStream, LongStream, etc, para generar un flujo de datos primitivos a continuación se presenta un ejemplo de forma imperativa y utilizando programación funcional:

Imperativa

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

Funcional

```
IntStream.range(0, 10).forEach(System.out::print);
```

Máximos y mínimos

A continuación se presenta una solución común para obtener el valor mínimo de una lista utilizando programación imperativa:

```
public static void main(String []args) {
    List<Integer>numbers= Arrays.asList(7, 2, 3, 100, 200, 300, 400, 5, 1);
    int min=numbers.get(0);
    for(Integer value:numbers) {
        if(value<min) {
            min=value;
        }
    }
    System.out.println(min);
}
```

A continuación se presenta la solución equivalente utilizando streams:

```
public static void main(String[]args) {
    List<Integer>numbers= Arrays.asList(7, 2, 3, 100, 200, 300, 400, 5, 1);
    Integer minValue= numbers.stream().
    min(Comparator.naturalOrder()).get();
    System.out.println(minValue);
}
```

Para calcular el valor mínimo solo se reemplazará min por max en la línea anterior.

Remover datos duplicados

Remover datos duplicados de una lista puede ser una tarea complicada, pero hacerlo con streams es muy simple, a continuación dos formas simples de hacerlo:

1. Utilizando distinct

```
public static void main(String[] args) {
    List<Integer> numbers= Arrays.asList(7, 7, 7, 7, 2, 2, 2, 3, 3, 3, 3, 100, 100, 200, 200);

    numbers=numbers.stream().distinct().
    collect(Collectors.toList());

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

2. Utilizando un set

```
public static void main(String []args) {
    List<Integer> numbers= Arrays.asList(7, 7, 7, 7, 2, 2, 2, 3, 3, 3, 3, 100, 100, 200, 200);
    Set<Integer>nums=
    numbers.stream().collect(Collectors.toSet());
    System.out.println(nums);
}
```

Transformaciones utilizando map

Puedes utilizar map cuando desees transformar de un objeto a otro, por ejemplo, si obtienes de una base de datos un objeto llamado **Persona** y lo quieres transformar a uno de tipo **String** puedes hacer lo siguiente:

```
public static void main(String []args) {
    List<Persona>personas=
    Arrays.asList(new Persona("Alex","Lopex"));

    List<String>nombres=personas.stream().
    map(p->p.getNombre()).
    collect(Collectors.toList());
    for(String nombre:nombres) {
        System.out.println(nombre);
    }
}
```

Uso de métodos static en interfaces

A partir de la versión 8 de Java puedes colocar métodos static en las interfaces como se muestra a continuación:

```
interface Follower{
    static String getName() {
        return "raidentrance";
    }
}
```

Los métodos static definidos en las interfaces no pueden ser sobreescritos.

Métodos default en las interfaces

Otro cambio importante al trabajar con interfaces es que ahora puedes incluir métodos con cuerpo, a estos los llamaremos métodos default, a continuación un ejemplo:

```
interface Follower{
    static String getName() {
    return"raidentrance";
    }

    default void follow() {
    System.out.println("Default follow impl");
    }
}
```

El método follow es un método con una implementación por default, a diferencia del métod getName que es static, el método follow puede ser sobreescrito sin nungún problema, consulta la interface **Comparator** para ver una interfaz que hace uso de métodos default.

Method reference

Una funcionalidad importante que se agregó en Java 8 es method reference y es utilizada para simplificar la ejecución de un método dentro de un lambda, esto puede ser de los siguientes modos:

- Referencias a métodos static
- Referencias a métodos de instancia
- Referencias a métodos de una clase
- Referencias a un constructor

A continuación se presentan algunos ejemplos.

Method reference por méodo de instancia

En este ejemplo se explicará como utilizar method reference con un método de instancia:

```
public static void main(String []args) {
    List<String>names=
    Arrays.asList("Alex","Juan","Pedro","raidentrance");

    names.stream().forEach(System.out::println);
}
```

Method reference static

En este ejemplo se explicará como utilizar method reference con un método static:

```
class StringUtils{
    static boolean isUpperCase(String cad) {
        return cad.toUpperCase().equals(cad);
    }
}
```

Utilizando programación imperativa:

```
public static void main(String []args) {
    List<String>names=
    Arrays.asList
("Alex","Juan","Pedro","raidentrance","PANCHO");

    names.stream().
    filter(cad->StringUtils.isUpperCase(cad))
    .forEach(System.out::println);
}
```

Utilizando static method reference:

```
public static void main(String[]args) {
    List<String>names=    Arrays.asList
("Alex","Juan","Pedro","raidentrance","PANCHO");

    names.stream().
    filter(StringUtils::isUpperCase)
    .forEach(System.out::println);

}
```

Method reference con constructores

Es posible utilizar constructores con method reference como se muestra a continuación:

```
class Person {
    private String name;

    public Person(String name) {
        this.name=name;
    }

    @Override
    public String toString() {
      return "Person [name="+name+"]";
    }
}

names.stream().filter(StringUtils::isUpperCase).
map(Person::new).forEach(System.out::println);
```

www.twitter.com/devs4j

www.facebook.com/devs4j

www.devs4j.com

Ejercicios

Sección 3 Primeros pasos

1. Realiza un programa capaz de calcular el **área** de un cuadrado, círculo, rectangulo y triangulo.

2. Toma el ejercicio anterior y combiertelo en un programa **ofuscado**, un programa ofuscado es un programa del cual no se entiende su funcionalidad a simple vista, haz uso de los conocimientos que tienes de identificadores para que sea algo difícil de entender.

Sección 4 Control de flujo

1. Realiza un programa que imprima las tablas de multiplicar del 1 al 20, debe existir un espacio entre una y otra para identificarlas.

2. Modifica el programa anterior para que solo se impriman las tablas de multiplicar de números pares, para identificar si los números son pares puedes utilizar el operador módulo con 2 como se muestra a continuación:

```
int x= 2;


if ( x %  2 == 0) {
 System.out.println("Es par");
}
```

3. Crea un programa utilizando ciclos que imprima el factorial de un número, se calcula multiplicando todos los números a partir del número indicado hasta 1 y se representa con el simbolo **!**, a continuación un ejemplo:

4! = 4 x 3 x 2 x 1 = 24

7! = 7 × 6 × 5 × 4 × 3 × 2 × 1 = 5040

1! = 1

El factorial de 0 es 1.

4. Crea un progama que imprima las siguientes figuras en la pantalla:

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

*
**
***
****
*****


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

Sección 5 Programación orientada a objetos

1. Realiza un diagrama de clases que represente mascotas.

2. Crea las clases definidas en el ejercicio anterior, cada una deberá tener más de un constructor y el código que se utiliza para inicializar no se deberá repetir.

3. Haz debugging sobre las clases del ejercicio anterior para que analices el paso a paso de la ejecución.

Sección 6 Herencia

1. Plantea 5 escenarios que hagan uso de herencia, recuerda que deben pasar la regla IS-A o ES-UN.

2. Utiliza el diagrama anterior y haz un programa que lo represente. Cada clase debe tener más de un constructor y todo debe funcionar correctamente, haz uso de la palabra reservada super.

3. Sobreescribe al menos uno de los métodos del ejercicio anterior.

Sección 8 Encapsulamiento y polimorfismo

1. Crea las clases necesarias para representar a un automovil, recuerda que debes aplicar los conceptos de polimorfismo y encapsulamiento.

Sección 9 y 10 Temas generales

1. Crear un arreglo de cada tipo de dato boolean, byte, short, int, char,float, long y double.

2. Crear un arreglo irregular en forma de triangulo.

3. Crear un arreglo de tipo Mascota y llenarlo con objetos de diferentes subtipos

4. Realizar 2 arreglos bidimensionales de las mismas dimensiones y sumarlos.

Sección 11 Colecciones

1. Crea una lista de mascotas modificando el ejemplo realizado en la sesión anterior.

2.Crea una implementación propia de una lista ligada, te recomendamos crear una clase Nodo que tenga como atributos una referencia al nodo siguiente y al anterior, debes soportar las operaciones agregar elemento, borrar y buscar.

3. Crea una clase que administre los elementos de una lista y te permita agregar, borrar y buscar un elemento. El único requisito es que la lista no permita datos duplicados.

4.Crea una implementación de alguna de las colecciones para esto debes crear una clase que implemente una de las interfaces List, Set, Map, Queue, etc.

Programación con Hilos

1. Crea un programa donde un hilo haga una cuenta regresiva dado un número, el tiempo que debe esperar el hilo debe ser aleatorio. Inicia 10 hilos y revisa que hilo termino primero, asegurate de colocarle un nombre a cada uno de ellos.

2. Modifica el programa anterior para que todos los hilos esperen el mismo tiempo asignale diferentes prioridades a cada uno de ellos y lanza 30 hilos, revisa si el cambio de prioridad generó algún cambio.

3. Simula un programa que haga transacciones bancarias donde se tengan que llevar a cabo las siguientes operaciones al retirar dinero:

- Validar el monto recibido
- Consultar saldo en la cuenta y validar si es suficiente para retirar el monto
- Restar el monto al saldo
- devolver el monto a quien invoco el método
- Imprimir que se realizó el retiro con exito

Ejecuta las operaciones sobre la cuenta con varios hilos sin utilizar sincronización te recomendamos hacer varios retiros, después hazlo de nuevo con sincronización.

4. Programa el algoritmo de los hilos fumadores, debes de tener 3 hilos que son los fumadores ellos necesitan 3 cosas para fumar, papel, tabaco y cerillos. A demás debes tener 3 hilos productores cada uno producirá un material diferente una vez que se encuentren los 3 materiales notifica a los hilos que pueden ir por sus materiales para fumar, si no hay materiales los hilos fumandores se deben ir a un estado waiting

Manejo de archivos y flujos

1. Programa el comando dir para listar los archivos en un directorio especificado

2. Programa el comando cp para copiar un archivo de un lugar a otro

3. Programa el comando mv para mover un archivo de un lugar a otro.

Sockets

1. Crea un programa que a través de un socket envíe un mensaje depende del mensaje el servidor imprimirá iniciando, reiniciando o apagando.

2. Crea un programa donde el cliente pueda enviar un archivo al servidor

3. Crea un programa donde el cliente pueda enviar un objeto llamado HTTPRequest que contenga los siguientes atributos:

- Body String
- Headers Map<String,String>

El servidor deberá responder un mensaje con los siguientes atributos:
- Body String
- Headers Map<String,String>
- HttpStatus Integer

Puedes seleccionar la acción a realizar te recomendamos que sea algo simple como la suma de dos números.

Generales

1. Crea un programa que corra en un servidor que tenga una lista de personas, el cliente debe ser capaz de leer las personas, enviar una persona nueva y modificar sus datos.

Todo se debe hacer de forma remota a través de sockets, si lo deseas puedes escribir un comando salir que al ejecutarlo se termine el proceso del lado del cliente y del servidor.

Al terminar el proceso del lado del servidor la lista se debe guardar en un archivo como objetos.

Java 9 - Modularidad

DEVS4J
Hecho por Devs para Devs

Java modular (Jigsaw)

El JSR(Java Specification Request) 376 - Java platform module system provee:

- Simplifica la construcción y el mantenimiento de aplicaciones grandes y complejas
- Mejora la seguridad y el mantenimiento del JDK (Modularizandolo)
- Mejora el performance de las aplicaciones
- Realiza un encapsulamiento más fuerte
- Simplifica su escalamiento hacia abajo para ser utilizado en dispositivios más pequeños

Con esto tendremos una plataforma más escalable con un mejor performance.

Más información en https://openjdk.java.net/projects/jigsaw/.

JEP's

Para conseguir el proyecto Jigsaw se crearon los siguientes JEP's (**JDK Enhancement proposal**):

- 200 - Java modular JDK
- 201 - Modular Source code
- 220 - Modular runtime images
- 260 - Encapsulate most of the internal apis
- 261 - Module system
- 282 - Jlink: The java linker

Crear módulos

Un módulo es básicamente un paquete de paquetes organizado por un archivo descriptor, a continuación se presenta un ejemplo donde se utiliza modularidad:

```
package com.devs4j.math.calculation;


import com.devs4j.math.operations.CalculatorProcessor;

public class ScientificCalculator{
    private CalculatorProcessor processor=new
CalculatorProcessor();

      public int sum(int x,int y) {
            return processor.sum(x,y);
      }
}
```

CientificCalcularor utiliza la clase CalcularorProcessor que se muestra a continuación:

```
package com.devs4j.math.operations;

public class CalculatorProcessor {
      public int sum(int x,int y) {
            return x+y;
      }
}
```

Se puede observar que se tienen los siguientes 2 paquetes:

```
package com.devs4j.math.calculation;
package com.devs4j.math.operations;
```

Calculation define la interfaz a compartir con otros proyectos y operations contiene los detalles de implementación, por esto solo deberíamos exponer el paquete calculation dado que operations solo se utiliza internamente, para hacerlo crearemos un archivo llamado **module-info.java** con la siguiente información:

```
module com.devs4j.math {
      exports com.devs4j.math.calculation;
}
```

El archivo module info define que solo el paquete calculation se expodrá y al no incluir el paquete operations se mantendrá como un detalle de implementación.

Uso de módulos

Una vez que se definió un módulo el siguiente paso es utilizarlo, para esto crearemos un nuevo módulo con la siguiente clase:

```
package com.devs4j.calculator;


import java.util.ArrayList;
import java.util.List;


import com.devs4j.math.calculation.CientificCalculator;


public class Devs4jTestCalculator {
    public static void main(String []args) {
     CientificCalculator calc=new CientificCalculator();
     System.out.println(calc.sum(10, 20));
     List<String>names=newArrayList<>();
     System.out.println(names);
    }
}
```

Para que el código anterior funcione sin ningún problema se deberá definir lo siguiente en el archivo **module-info.java**:

```
module com.devs4j.calculation {
      requires java.base;
      requires com.devs4j.math;
}
```

El archivo anterior define que nuestro proyecto requiere del paquete com.devs4j.math para funcionar, la declaración **requires java.base**; no es necesaria dado que se hace de manera implícita pero es la que permite que utilicemos clases como ArrayList.

Archivo module-info.java

Cuando se crea un nuevo módulo se debe crear un archivo descriptor llamado **module-info.java** el cual puede definir lo siguiente:

- **Nombre** : Nombre del módulo
- **Dependencies** : Lista de dependencias del módulo
- **Paquetes publicos**: Lista de paquetes accesibles fuera del módulo
- **Servicios ofrecidos**: Lista de servicios que pueden consumir otros módulos
- **Servicios a consumir**: Lista de servicios a consumir de otros módulos
- **Reflection permissions**:Permite definir de forma explícita el nivel de acceso que se tendrá al utilizar reflection.

La convención de nombres de los módulos es similar a la de los paquetes.

Por default todos los paquetes de un módulo son privados.

Tipos de módulos

A continuación se listan los tipos de módulos disponibles:

- **System modules**: Lista de módulos que contienen los módulos del JDK puedes acceder a ellos utilizando el comando list modules.
- **Application modules**: Módulos que crearemos y nombraremos durante la construcción de nuestras aplicaciones.
- **Automatic modules**: Módulos que estarán disponibles al incluir un archivo jar. Por default tendran completo acceso de lectura a todos los módulos agregados al path.
- **Unnamed modules**: Son utilizados para tener compatibilidad con versiones anteriores de java se utilizan cuando no se incluye ningún tipo de módulo.

Los módulos pueden ser distribuidos a través de archivos jar(Solo podemos tener un módulo por jar).

Declaración de módulos

Antes de declarar los módulos del sistema utilizando **java --list-modules**, la declaración de un módulo se hace creando un archivo llamado module-info.javan incluyendo el nombre del módulo como se muestra a continuación:

```
module com.devs4j.calculation {
      ...
}
```

Requires

Se utiliza para definir las dependencias de nuestro módulo a continuación un ejemplo:

```
module com.devs4j.calculation {
      requires com.devs4j.math;
}
```

Todos los paquetes exportados por el módulo definido estarán disponibles para su uso.

Requires static

Define una dependencia que es opcional en tiempo de ejecución pero que es necesaria en tiempo de compilación.

```
module com.devs4j.calculation {
      requires static com.devs4j.math;
}
```

Requires transitive

Cuando dependemos de un modulo que a su vez tiene dependencias utilizaremos requires transitive como se muestra a continuación:

```
module com.devs4j.calculation {
      requires transitive com.devs4j.math;
}
```

Exports

Por default un módulo no expone ninguno de los paquetes, por motivos de encapsulamiento fuerte(Strong encapsulation) debemos definir de forma explícita los paquetes que serán accesibles como se muestra a continuación:

```
module com.devs4j.math {
      exports com.devs4j.math.calculation;
}
```

Puedes utilizar exports ... to ... para restringir los módulos que pueden importar el paquete especificado.

Java 9 - Modularidad

DEVS4J
Hecho por Devs para Devs

Produce / Consume services

Los módulos pueden producir y consumir servicios, más que solo paquetes, un servicio es una interfaz que puede tener múltiples implementaciones, a continuación algunos puntos importantes:

- **provides** <service interface> **with** <classes> especifica que el módulo provee una o más implementaciones que pueden ser descubiertas de forma dinámica por el consumer.
- **uses** <service interface> especifica una interfaz o clase abstracta que se desea consumir.

A continuación un ejemplo:

Service

```
package com.devs4j.service.auth;


public interface UserAuthentication {
	public boolean authenticate(String username, String password);
}
module com.devs4j.service {
	exports com.devs4j.service.auth;
}
```

Provider

```
module com.devs4j.provider {
	requires com.devs4j.service;
	provides com.devs4j.service.auth.UserAuthentication
	with UserDatabaseAuthenticator;
}
```

Application

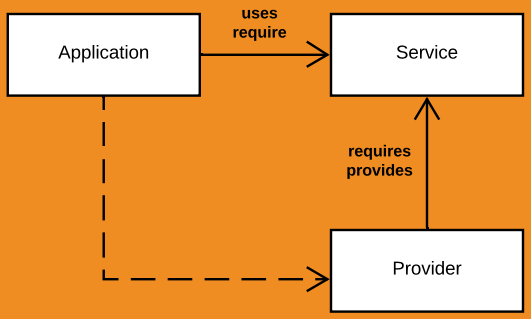
```
public class TestAuthenticationApplication {
	public static void main(String[] args) {
		ServiceLoader<UserAuthentication> service =
			ServiceLoader.load(UserAuthentication.class);

		UserAuthentication userAuthentication =
		service.findFirst().get();

		boolean authenticated =
	    userAuthentication.authenticate("raidentrance","udemy");

		System.out.println(authenticated);
	}
}

module com.devs4j.application {
	requires com.devs4j.service;
	uses UserAuthentication;
}
```



Reflection

Reflection es un api que te permite trabajar con clases, objetos, métodos y otros componentes del lenguaje como clases que podemos manipular a continuación un ejemplo:

```
package com.devs4j.reflection.service;

class ApplicationService {
	void sayHello() {
		System.out.println("Hello");
	}
}
```

```
package com.devs4j.reflection.user;


import java.lang.reflect.Constructor;
import java.lang.reflect.Method;


public class TestApplicationService {
public static void main(String []args)throws Exception
{
	Class<?>classObject=	Class.forName
	("com.devs4j.reflection.service.ApplicationService");

	Constructor<?>constructor=
	classObject.getDeclaredConstructor();

	constructor.setAccessible(true);
	Object instance=constructor.newInstance();

	Method[]methods=classObject.getDeclaredMethods();
	for(Methodmethod:methods) {
		System.out.printf("Method %s, is accessible
	\n",method.getName(),method.canAccess(instance));

		if(method.getName().equals("sayHello")) {
			method.setAccessible(true);
			method.invoke(instance);
		}
	}
}
}
```

Como se puede ver **ApplicationService** tiene un nivel de acceso default, esto evita que puedas utilizarla fuera del paquete **com.devs4j.reflection.service** sin embargo haciendo uso de reflection podemos tener acceso sin importar si la clase se encuentra fuera del paquete.

Open

Los módulos pueden permitir acceso a través de reflection utilizando open.

- **opens** <paquete> indica que se permitirá acceso a través de reflection
- **opens** <paquete> **to** <module> restringe la apertura de los paquetes a algún modulo en específico.

Opens funciona igual que export con la diferencia que permite acceso a los tipos no públicos a través de reflection.

Probando reflection con módulos

Es tiempo de combinar los dos ejemplos anteriores para entender el funcionamiento de opens, para esto se modificarán los siguientes componentes:

Service

```
package com.devs4j.reflection.app;


class ApplicationService {
	void sayHello() {
		System.out.println("Hello");
	}
}

module com.devs4j.service {
	exports com.devs4j.service.auth;
	exports com.devs4j.reflection.app;
}
```

Se puede observar que se uiliza exports por lo que el acceso a través de reflection será restringido.

Application

```
public class TestAuthenticationApplication {
 public static void main(String []args) {
	ServiceLoader<UserAuthentication>service=
	ServiceLoader.load(UserAuthentication.class);

	UserAuthenticationuserAuthentication=
	service.findFirst().get();

	boolean authenticated=	userAuthentication.
	authenticate("raidentrance","udemy");

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

Se puede observar que se uiliza exports por lo que el acceso a través de reflection será restringido.

Al ejecutar el código anterior obtendremos la siguiente excepción:

```
Exception in thread "main"
java.lang.reflect.InaccessibleObjectException:
Unable to make
com.devs4j.reflection.app.ApplicationService()
accessible: module com.devs4j.service does not
"opens com.devs4j.reflection.app" to module
com.devs4j.application
```

Para permitir el acceso a través de reflection debemos modificar la definición del módulo en service como se muestra a continuación:

```
module com.devs4j.service {
	exports com.devs4j.service.auth;
	opens com.devs4j.reflection.app;
}
```

Una vez realizado el cambio la ejecución del método debe ser correcta

Java 9

DEVS4J
Hecho por Devs para Devs

Try with resources

Try with resources apareció en la versión 7 de java con la siguiente estructura:

```
try(InputStreamReader isr=new InputStreamReader(System.in);){

}
```

A partir de la versión 9 de Java podremos realizar las declaraciones fuera del try como se muestra a continuación:

```
InputStreamReader isr = new InputStreamReader(System.in);
try(isr) {

}
```

Extensión al operador diamante

En Java 7 se agregó el soporte para utilizar el operador diamante como se muestra a continuación:

```
List<Integer>list=new ArrayList<>();
```

Una de las restricciones que se tenía es que el operador diamante no se podía aplicar a clases anónimas y se debía hacer del siguiente modo:

```
interface Foo<T>{}

public static void main(String []args) {
      Foo<Integer>foo=new Foo<Integer>() {
      };
}
```

A partir de Java 9 ya puedes utilizar el operador diamante con clases abstractas como se muestra a cotinuación:

```
interface Foo<T> {
}

public static void main(String []args) {
      Foo<Integer> foo=new Foo<>() {
      };
}
```

Métodos privados en interfaces

A partir de la versión Java 9 puedes definir métodos privados con el fin de que sean utilizados por los métodos default, como se muestra a continuación:

```
interface Bar {
      private void foo() {
      }

      private static void foo2() {
      }

      default void foo3() {
            foo();
            foo2();
      }
}
```

Es importante mencionar que los métodos abstractos deben seguir siendo public.

JShell

JShell (Java Shell) es una herramienta interactiva para el aprendizaje del lenguaje java, se utiliza para evaluar declaraciones, sentencias y expresiones, estas inmediatamente mostrarán los resultados.

Cuando utilizar JShell

Puedes utilizar JShell para escribir elementos una vez e inmediatamente obtener resultados.

Configuración

Para poder ejecutar el comando **jshell** y utilizarlo debes instalar el JDK de java y agregar a las variables de entorno **java-home/jdk-9/bin.**

Una vez que se configura la variable de entorno puedes ejecutar el comando jshell como se muestra a continuación:

```
% jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro
jshell>
```

Para salir de jshell utiliza el comando **/exit** como se muestra a continuación:

```
jshell> /exit
|  Goodbye
```

Snippets

Jshell acepta sentencias java; variables, métodos, clases, imports y expresiones. Estas piezas de código son conocidas como snippets.

Declaración de variables

```
jshell> int x = 45
x ==> 45
|  created variable x : int
```

Evaluación de expresiones

```
jshell> 2 + 2
$3 ==> 4
|  created scratch variable $3 : int
```

```
jshell> String twice(String s) {
 ...>    return s + s;
...> }
|  created method twice(String)
jshell> twice("Ocean")
$5 ==> "OceanOcean"
|  created scratch variable $5 : String
```

Cambiando la definición

```
jshell> String twice(String s) {
 ...>   return "Twice:" + s;
...> }
|  modified method twice(String)
jshell> twice("thing")
$7 ==> "Twice:thing"
|  created scratch variable $7 : String
```

Commands

Puedes utilizar comandos para controlar el ambiente y mostrar información a continuación algunos ejemplos:

```
jshell> /vars
|    int x = 45
|    int $3 = 4
|    String $5 = "OceanOcean"

jshell> /methods
|    twice (String)String

jshell> /list
1 : System.out.println("Hi");
2 : int x = 45;    3 : 2 + 2
4 : String twice(String s) {return s + s;}
5 : twice("Ocean")
```

Puedes escribir / y utilizar **tab** para obtener los comandos disponibles.

Puedes utilizar abreviaciones como /l, /se fe v (set feedback verbose)

Feedback

Puedes utilizar el feedback mode para determinar las respuestas y otras interacciones con jshell.

Los feedbacks disponibles son:

- verbose
- normal
- concise
- silent

Para cambiar el tipo de feedback utiliza el siguiente comando:

```
/set feedback verbose
```

Prueba los diferentes tipos de feedback para ver los diferentes tipos de interacción que puedes tener.

Scripting

Puedes escribir scripts en archivos con extensión **.jsh**, pueden ser generados como salidas de las entradas del shell como se muestra a continuación:

```
jshell> /save mysnippets.jsh
jshell> /save -history myhistory.jsh
jshell> /save -start mystartup.jsh
```

Puedes ejecutarlos como se muestra a continuación:

```
% jshell mysnippets.jsh
```

Para cargar un script puedes utilizar /open como se muestra a continuación:

```
jshell> /open PRINTING
```