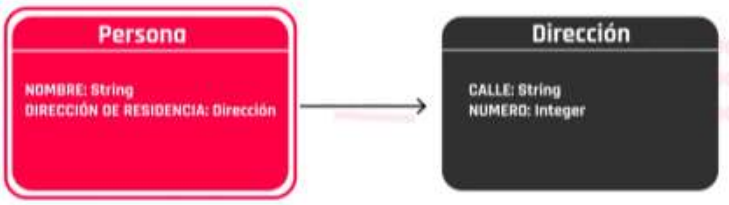
Programación Orientada a Objetos
Profesor: Rodolfo Baspineiro

Módulo 2: Programación Orientada a Objetos en Java

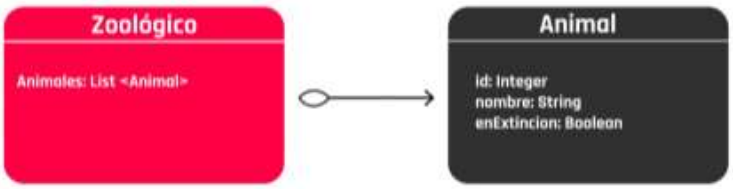
Sesión 7: Relaciones entre clases
Noviembre 1 de 2021

Tipos de relaciones:
- Asociación
- Agregación
- Composición
- Generalización
- Especialización

Asociación:
Relación unilateral. "A tiene B".



Agregación.
"A es parte de B"



Composición
"A está compuesta de B"

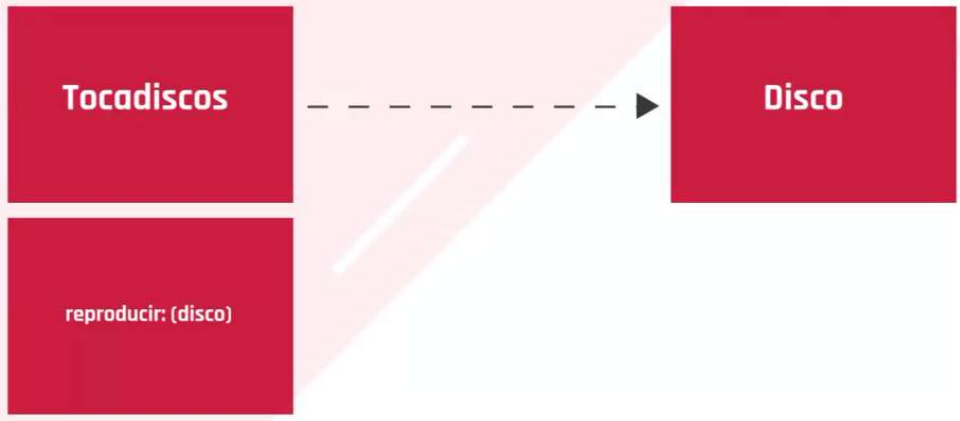
.

En la composición, la relación es indisociable, una clase no puede existir sin la existencia de la otra. En la agregación, las clases pueden seguir existiendo aunque se rompa la relación entre ellas.

Relación de Asociación

Ej: "Un dueño tiene una mascota".

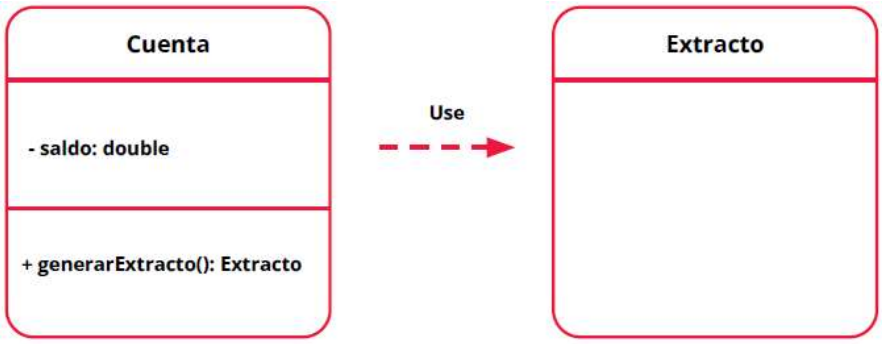
Relación de uso o dependencia
Ej: "Un tocadiscos reproduce un disco". En este caso, el disco no es un atributo del tocadiscos, sino que aparece como parámetro en uno de sus métodos. Se trata de una colaboración temporal.



Navegación
Un objeto puede acceder a otro porque contiene referencias específicas a él. En general, esto no se cumple en el sentido contrario.

Cardinalidad o Multiplicidad
Especifica el número de instancias de una clase que puede estar relacionadas con una única instancia de una clase asociada.

Relación de uso
No hay una referencia de una clase a la otra; la relación se da porque hay algún método que devuelve o recibe como parámetro una variable que es del tipo de la otra clase.



Relación de Agregación:
Existe una relación entre los agregados y el todo, pero los componentes pueden existir aunque el todo fuese destruido.

Relación de Composición:
Es un tipo de agregación más fuerte, donde la parte no tiene sentido sin el todo.

En el ejemplo, si existe un empleado, tiene que existir una empresa donde ese empleado trabaje.

Implementación en Java

Relación 1 a 1. Auto → Motor

```
public class Auto {
        private Motor motor; // El motor se incluye como atributo
        private int velocidadMaxima;
        private String marca;
}

public class Motor {
        private String fabricante;
        private String potencia;
}
```

En el diagrama UML no se muestra el motor como atributo, ya que se entiende a partir de la relación y sería información redundante.

Relación 1 a muchos. Auto → Rueda

```
public class Auto {
	private Rueda[] rueda; // Se utiliza un array o colección
	private int velocidadMaxima;
	private String marca;
}

public class Rueda {
	private int tamano;
}
```

Relación de composición. Televisor *→ Pantalla

El constructor de la clase Televisor invoca al constructor de la clase Pantalla. De este forma una instancia de la clase Pantalla se crea siempre asociada a una instancia de Televisor.

```
public class Televisor {
	private String marca;
	private Pantalla pantalla;

	public Televisor() {
		pantalla = new Pantalla(); // Al crear una instancia de Televisor se crea una instancia de Pantalla
	}
}

public class Pantalla {
	private int pulgadas;
}
```

Cuando se crea una instancia de un objeto que tiene como atributo un objeto, este también debe estar instanciado, o instanciarse en el constructor.
Ejemplo:

```
public class Auto {
	private String patente;
	private String marca;
	private String modelo;
	private Motor motor;
```

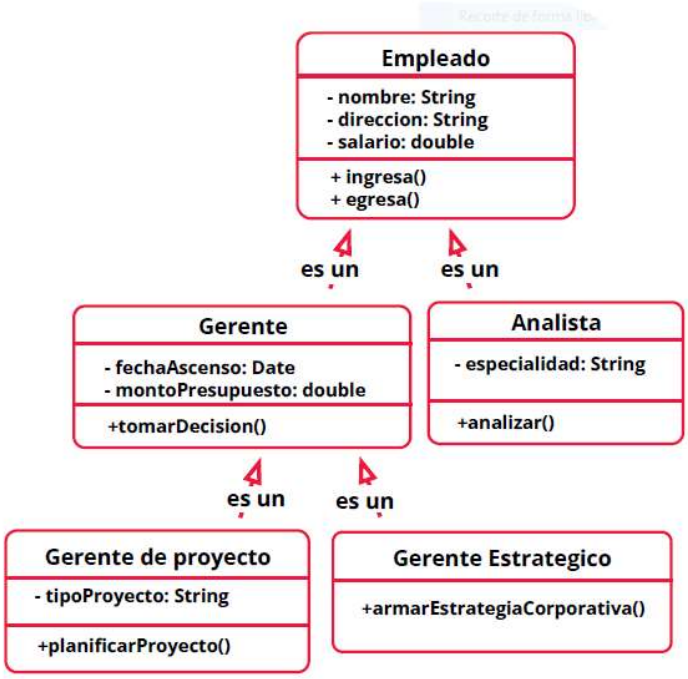
```
	public Auto(String patente, String marca, String modelo) {
		this.patente = patente;
		this.marca = marca;
		this.modelo = modelo;
		motor = new Motor(); // Se instancia el atributo, aunque 		sea con un objeto vacío.
	}
```

Sesión 8: Herencia en UML

Noviembre 3 de 2021

Herencia

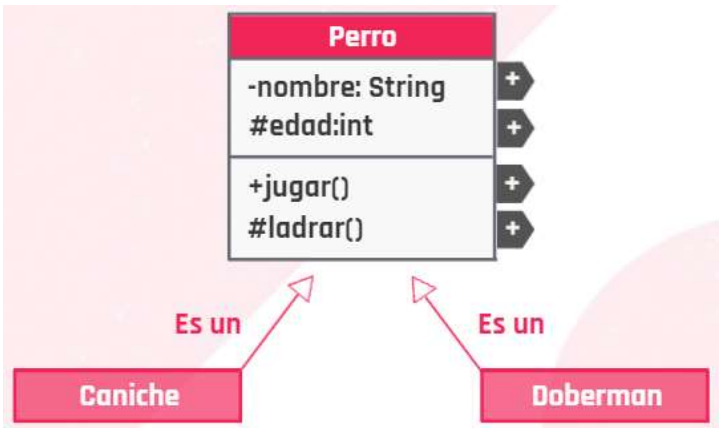
Es un ordenamiento entre clases que define una relación "es un". La Herencia favorece la reutilización de código.
Una clase que hereda de otra, suma a sus propios atributos y responsabilidades los de la clase a la cual hereda.



Herencia múltiple

Se establece cuando una clase hereda de varias otras clases, en este caso, la clase hija hereda atributos y responsabilidades de los diferentes padres.

Encapsulamiento y herencia



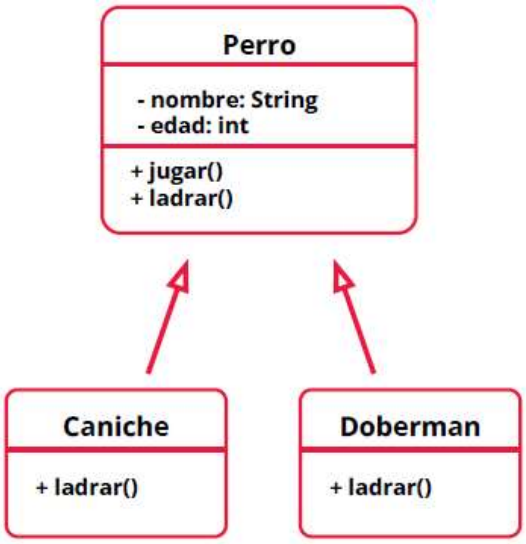
.

Con la herencia aparece el modificador de visibilidad *protected*, que en los diagramas UML se especifica con el "#", que permite tener una visibilidad intermedia del atributo o método declarado como tal. Es decir, es privado para otras clases, pero público para las clases hijas.

Firma de un método
Es la definición completa de un método, es decir, su nombre y la cantidad, tipo y orden de sus parámetros.
El valor que devuelve un método y los modificadores de visibilidad no forman parte de la firma.

Sobrecarga de métodos
Consiste en tener en una misma clase dos o más métodos con el mismo nombre y cuyo comportamiento sea diferente. Es factible siempre que los métodos tengan diferente firma.

Sobreescritura de métodos
Consiste en sobreescribir un método de la superclase para que se comporte diferente en la subclase. Requiere que los métodos tengan la misma firma.



Sesión 10. Herencia en Java

Noviembre 8 de 2021

```
public class Persona {
	private String nombre;
	private String dni;

	public Persona(String nombre, String dni) {
		this.nombre = nombre;
		this.dni = dni;
	}
}

public class Empleado extends Persona { // La palabra reservada  extends indica la relación de herencia
	private double sueldo;
	private double descuento;
	private String legajo;

	public Empleado(String nombre, String dni, String legajo) {
		super(nombre, dni); // Se invoca a la superclase para instanciarla asignando los atributos nombre y dni
		this.legajo = legajo;
		this.sueldo = 30000;
	}
}
```

Sobrecarga y sobreescritura

Sobrecargar un método

Es posible sobrecargar un método si varía su firma.

```
public class Empleado{
	private String nombre;
	private String legajo;
	protected double sueldo;
	protected double descuentos;

	public double calcularSueldo(){
		return sueldo-descuentos;
	}
	public double calcularSueldo(double premio){
		return sueldo-descuentos + premio;
	}
}
```

```
public class Main {
	public static void main(String[] args) {
		Empleado miEmpleado = new Empleado(“Juan”, “1111”);
		System.out.println(“Sueldo a cobrar: ” +
	miEmpleado.calcularSueldo());
		System.out.println(“Sueldo a cobrar: ” +
	miEmpleado.calcularSueldo(5000)); // El método que se ejecuta  depende de los parámetros que se pasan.
	}
}
```

Sobreescribir un método

La anotación @override anula la definición anterior del método.

```
public class Vendedor extends Empleado{
	private int comision;
	private double importeVentas;

	@Override // Sobreescribe el método existente
	public double calcularSueldo(){
		return sueldo-descuentos + importeVentas/100*comision;
	}
	@Override // Sobreescribe el método existente
	public double calcularSueldo(double premio){
		return sueldo-descuentos + premio+ importeVentas/100*comision;
	}
}
```

La clase Object

Sobreescribir .toString()

```
public class Empleado{
	private String nombre;
	private String legajo;
	protected double sueldo;
	protected double descuentos;

	@Override // Sobreescribe el método toString() para que  muestre información del objeto
	public String toString(){
		return “Nombre: ” + nombre + “\n” +
		“Legajo: ” + legajo;
	}
}
```

Sobreescribir .hashCode()

Incomprensible. ¿Para qué se sobreescribe?

Sobreescribir .equals(Object o)

En el ejemplo se considera que dos Empleados son iguales si sus legajos son iguales:

```
public class Empleado{
	private String nombre;
	private String legajo;
	private double sueldo;
	private double descuentos;

	@Override
	public boolean equals(Object o){
		if (o==null)
			return false; // Primero se verifica que el objeto no sea nulo
		if (!(o instanceof Empleado))
			return false; // A continuación se verifica que o sea una instancia de empleado
		else{
			Empleado emp=(Empleado)o;
			return this.getLegajo().equals(emp.getLegajo());
			// Finalmente se comprueba que los legajos
	coinciden. Es necesario castear, ya que o es un Object, es decir, 'no sabe' que es un Empleado.
		}
	}
}
```

Otra forma de hacerlo es usar el método getClass() en lugar de instanceOf():

```
public class Empleado{
	private String nombre;
	private String legajo;
	private double sueldo;
	private double descuentos;

	@Override
	public boolean equals(Object o){
		if (o==null)
			return false; // Primero se verifica que el objeto no sea nulo
		if (!(this.getClass()==o.getClass()))
			return false; // A continuación se verifica que o sea una instancia de empleado
		else{
```

```
                    Empleado emp=(Empleado)o;
                    return this.getLegajo().equals(emp.getLegajo());
                    // Finalmente se comprueba que los legajos
        coinciden. Es necesario castear, ya que o es un                    Object, es decir,
'no sabe' que es un Empleado.
                }
        }
}
```

Sincrónico

- ¿Para qué se sobreescribe el método .hashCode?
- ¿en qué casos se usa protected? ¿Una subclase requiere acceder a los atributos de su superclase a través de getters y setters, o los atributos deberían definirse como protected?

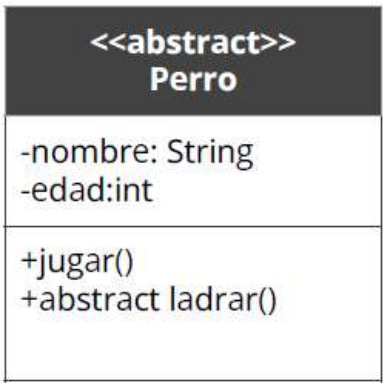
Sesión 11. Clases abstractas

Noviembre 10 de 2021

Clases abstractas

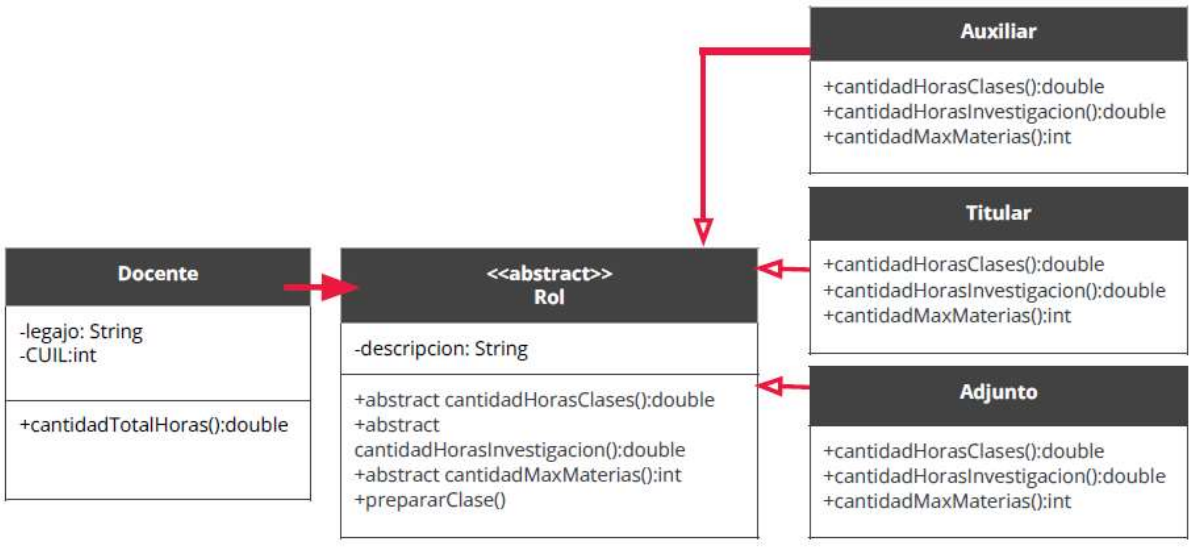
- No pueden ser instanciadas
- Permiten declarar métodos, mas no implementarlos. Estos métodos abstractos deben ser implementados en las subclases de la clase abstracta.

Clase abstracta en el diagrama UML:



Para indicar que la clase o el método son absctractos, se utiliza la palabra **abstract**. De manera alternativa para indicar que la clase es abstracta, se podría escribir el nombre de la clase en cursiva.

Ejemplo: Rol abstracto



Martin Fowler describió a estaproblemática de roles “Role Object” como un patrón de diseño para modelar roles, siendo una buena práctica a la hora de modelar esta problemática.

Clases abstractas en Java:

```
public abstract class Perro{
        private String nombre; // La clase abstracta puede tener    atributos y métodos concretos

        public void setNombre(String nombre){
                this.nombre = nombre;
        }
        public String getNombre(){
                return nombre;
        }
        public abstract string ladrar(); // Los métodos abstractos no         tienen código asociado, es decir, no tienen cuerpo.
}


public class Doberman extends Perro{
        public string ladrar() {
                return “ladro como un dóberman GUAU!!!”;
        } // La subclase (Doberman) debe implementar todos los métodos abstractos definidos en la Superclase (Perro). Debe respetar el tipo, cantidad y orden de los parámetros.
}
```

@override

```
public class Perro {
        public String ladrar() {
                return “guau”;
        }
}

public class Doberman extends Perro{
        // La anotación @override pide al compilador validar que el         método tiene una firma igual a algún método de la superclase,        permitiendo detectar posibles errores.
        @Override
        public String ladrar() {
                return “GUAU GUAU!!!”;
        }
}
```

Polimorfismo

Dynamic binding:

```
Doberman perro = new Doberman();

// El lado izquierdo es una referencia a un tipo de terminado, una variable, pero no implica la creación de un objeto.

// En el lado derecho se crea una instancia (un objeto de la clase Doberman.

// La asignación = genera el binding entre la referencia y el objeto. En el ejemplo, tanto la referencia como el objeto son del mismo tipo; sin embargo, puede ser que los tipos no coincidan. En general, el binding se puede establecer si el tipo del objeto guarda una relación "es un" con el tipo de la referencia.

Perro perro = new Doberman(); // Válido.

Doberman perro = new Perro(); // Inválido.

Object perro = new Doberman(); // Válido.
```

Polimorfismo

Es la capacidad de un mismo objeto de comportarse como otro. En otras palabras, es la capacidad de un objeto de funcionar de diversas formas.

Ejemplo:

```
Perro p;
p = new Doberman(); // La referencia p se puede comportar como un Doberman
p.ladrar();
p = new Caniche(); // La misma referencia p al vincularse
dinámicamente con un Caniche, se comporta como un Caniche.
p.ladrar();
```

Casting

Ejemplo

```
Perro perro = new Doberman();
perro.ladrar();
((Doberman)perro).morderComoDoberman() // El casting permite que el objeto perro se comporte como instancia de la clase Doberman y se puedan invocar sus métodos.

Object perro = new Doberman();
((Perro)perro).ladrar();
((Doberman)perro).morderComoDoberman();
```

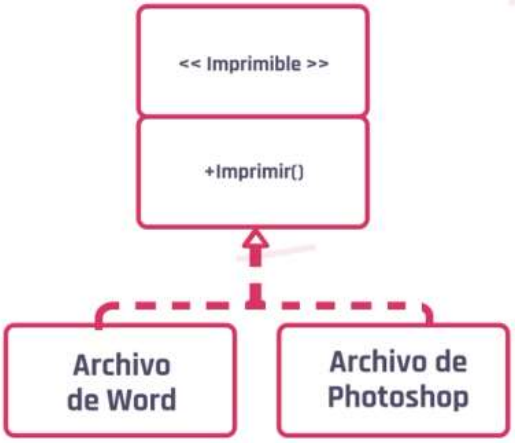
Clases abstractas vs clases concretas

Clases abstractas:
- Usualmente tienen implementación parcial o nula.
- No pueden ser instanciadas.
- Pueden contener métodos abstractos.
- Tienen que extenderse para que tenga sentido su existencia.

Clases concretas:
- Siempre tienen plena implementación de su comportamiento.
- Pueden ser instanciadas.
- No pueden contener métodos abstractos.

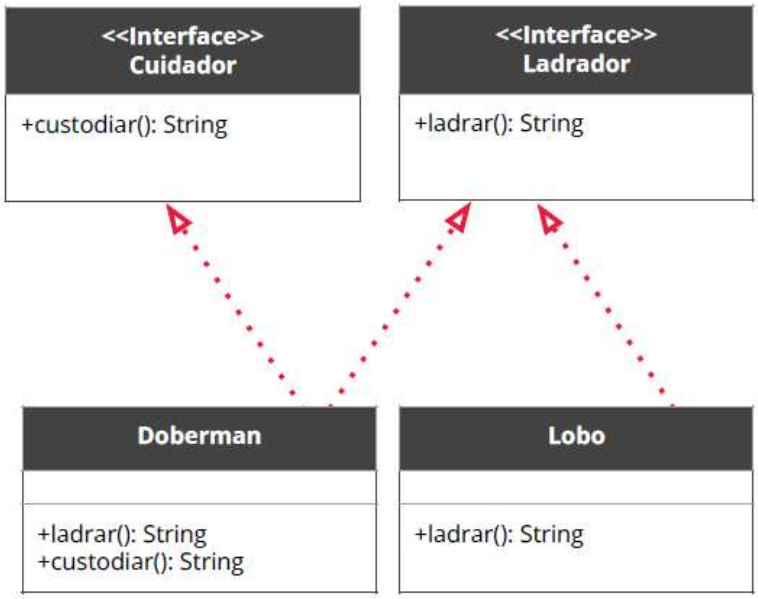
Sesión 13. Interfaces.
Noviembre 15 de 2021.

Interfaces:
- No tienen atributos
- En el diagrama UML se distinguen del resto de las clases escribiéndolas entre comillas españolas (<< Interfaz >>).
- No permiten crear instancias



- Las interfaces constituyen una solución al problema de la herencia múltiple
Las interfaces son relaciones del tipo "es un", similares a las clases abstractas. Todos sus métodos son abstractos, y cualquier clase que implemente una interfaz debe implementar todos sus métodos.

Una clase solo puede heredar de una clase, pero puede implementar varias interfaces:



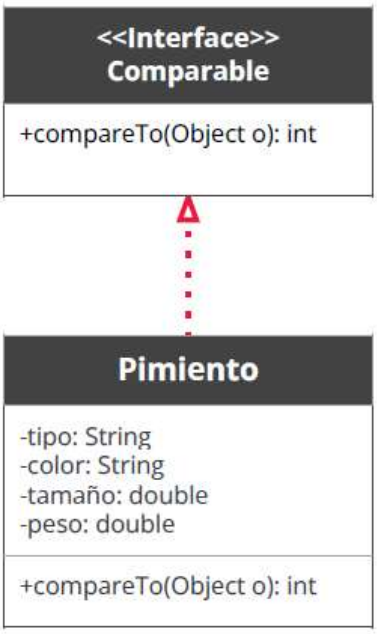
Las interfaces permiten realizar vinculación dinámica y por lo tanto polimorfismo.

Ejemplo de Implementación de Interfaces:

```
public interface Cuidador{
        public void String custodiar();
}

public interface Ladrador{
        public void String ladrar();
}

public class Doberman implements Cuidador, Ladrador{
        public void String custodiar(){
        return “estoy atento custodiando la casa”;
        }
        public void String ladrar(){
                return “Guau! Guau!”;
        }
 }

public class Lobo implements Ladrador{
        public void String ladrar(){
                return “guau! los lobos también ladramos”;
        }
}
```

Interface Comparable

Interfaz de Java que permite comparar objetos.



.

Ejemplo:

```
import java.lang.*;

public class Pimiento implements Comparable{
	private String tipo;
	private String color;
	private double tamano;
	private double peso;

	public Pimiento(){
	}

	public int compareTo(Object obj){
		Pimiento p2 = (Pimiento) obj;
		int respuesta = 0;
		if(this.getPeso() > p2.getPeso())
			respuesta = 1;
		if(this.getPeso() < p2.getPeso())
			respuesta = -1;
		return respuesta;
	}

	//Getters, setters, etc.
}
```

Sincrónico

Al definir métodos en las interfaces no es necesario especificar que son públicos y abstractos (se sobreentiende).

Ejemplo:

```
public Interface ImpuestoGravable {
	double gravar(double porcentaje);
}
```

Generalmente se define una gran cantidad de métodos en una interface.

SistemaArmas
SistemaArmasVoladoras
+ volar()
Tanque
Bombardero
Submarino
+ sumergirse()

Portaaviones
VehículoSeñuelo

Sesión 16. Colecciones

Noviembre 22 de 2021

Colecciones en Java:

Una colección representa un grupo de objetos (elementos)

Java proporciona la interface Collection, de la cual se extienden las interfaces Set y List. La diferencia entre ambas es que Set no admite elementos repetidos; por su parte List define una sucesión de elementos.
Java también proporciona la interface Map que permite almacenar parejeas clave-valor, donde las claves deben ser únicas. Map no extiende de Collection.

Tipos de Colecciones

ArrayList

Implementa la interface List. Almacena y permite el acceso a los elementos de forma secuencial.

Métodos:
.add(Object o)
.add(Object o, int pos)
.remove(Object o)
.remove(int pos)
.get(int pos)
.size()

LinkedList

Implementa la interface List. Es útil cuando se requiere hacer inserciones cerca de la mitad de la lista, pero es poco eficiente cuando solo se requiere agregar o acceder a los elementos, caso en que se recomienda usar ArrayList.

Métodos:
.add(Object o)
.add(Object o, int pos)
.remove(Object o)
.remove(int pos)
.get(int pos)
.size()

HashSet

Implementa la interface Set. No permite almacenar valores duplicados o nulos. Es la colección que presenta el mayor rendimiento, pero no garantiza el orden al recorrerla.

Métodos:

.add(Object o)
.remove(Object o)
.size()

LinkedHashSet
Implementa la interface Set. No admite valores nulos ni duplicados, pero los elementos se almacenan en el orden de inserción. Tiene menor rendimiento que HashSet.

Métodos:
.add(Object o)
.remove(Object o)
.size()

TreeSet
Implementa la interface Set, pero además hereda de la clase SortedSet, lo cual le permite almacenar los elementos de acuerdo con sus valores.

Métodos:
.add(Object o)
.remove(Object o)
.size()

HashMap
Implementa la interface Map. Los datos no se almacenan en el mismo orden de inserción.

Métodos:
.put(Object key, Object value)
.get(Object key)
.remove(Object key)
.size()

LinkedHashMap
Implementa la interface Map. A diferencia de HashMap, los elementos se almacenan en el orden de inserción.

Métodos:
.put(Object key, Object value)
.get(Object key)
.remove(Object key)
.size()

TreeMap
Implementa la interface Map. Los elementos se almacenan ordenadamente según la clave,

Métodos:
.put(Object key, Object value)
.get(Object key)
.remove(Object key)
.size()

Recorrer una colección

Ciclo for:

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

Ciclo while:

```
int i = 0;
while( i < nombres.size()){
	System.out.println(nombres.get(i));
	i++;
}
```

Los métodos anteriores solo se pueden emplear con ArrayList y LinkedList.

Iterator

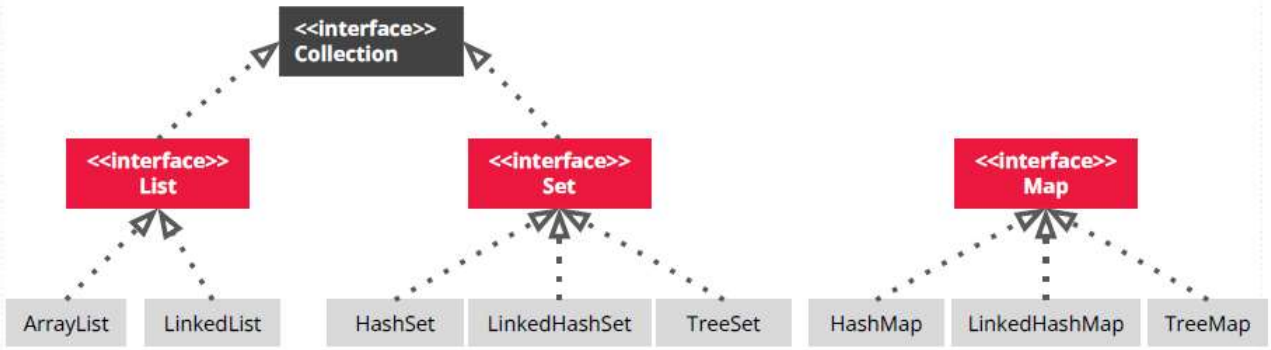
```
Iterator iterator = nombres.iterator();
while(iterator.hasNext()){
	System.out.println(iterator.next());
}
```

En Java las colecciones implementan la interface Iterable, lo que obliga a implementar el método iterator(). Iterator() devuelve un objeto del tipo Iterator, donde mediante los métodos hasNext() y next() se puede recorrer la colección.

For each

```
for(Object nombre: nombres){
	System.out.println(nombre);
}
```

Crear una colección



Al crear una colección o cualquier tipo de objeto, es una buena práctica que el tipo de la referencia sea lo más genérico posible:

```
List nombres = new ArrayList();
```

Generics. Programación paramétrica

```
public class Balde<T> {

	private T contenido;

	public Balde() {
	}

	public llenar(T contenido) {
		this.contenido = contenido;
	}

	public T obtenerContenido() {
		return contenido;
	}
}
```

El tipo de balde se define con posterioridad:

```
Agua a = new Agua();
Combustible c = new Combustible();
Balde<Agua> b = new Balde<>();
b.setContenido(a);
```

Al hacerlo de esta forma no es necesario realizar casteos para operar con el contenido.

```
//b.setContenido(c);  // Generaría error.
```

Colecciones paramétricas

Ejemplo:
Una empresa de transportes tiene dos tipos de vehículos: motos y camiones. Se dese almacenar estos vehículos en una lista.

Sin utilizar Generics:

```
List vehiculos = new ArrayList();

Moto moto = new Moto();
Camion camion = new Camion();

vehiculos.add(moto);
vehiculos.add(camion);
```

Para obtener algún elemento particular, se requeriría castear:

```
Moto moto = (Moto) vehiculos.get(0);
Camion camion = (Camion) vehiculos.get(1);
```

Para recorrer la lista de vehículos empleando un método de la superclase:

```
for(Object o :vehiculos) {
        System.out.println(((Vehiculo)o).estaDisponible());
}
```

Sin embargo, si se desea aplicar un método que pertenece a una de las subclases, se requiere verificar el tipo cono instanceof:

```
for(Object o :vehiculos) {
        if(o instanceof Camion)
                ((Camion)o).cargar(“papas”);
}
```

Utilizando Generics:

```
List<Camion> vehiculos = new ArrayList<Camion>();
```

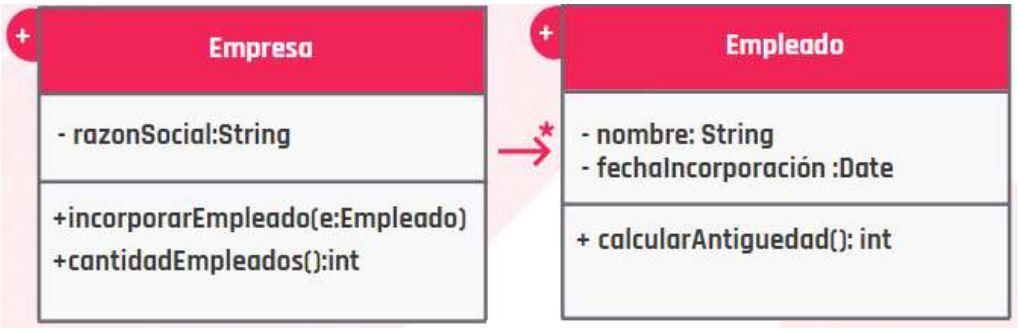
Al crear la lista se especifica el tipo.

De esta forma se puede utilizar el método de la subclase sin verificar el tipo con instanceof, debido a que los tipos no pueden mezclarse:

```
for(Camion o :vehiculos) {
        o.cargar("papas");
}
```

Relaciones 1 a muchos con colecciones:

Ejemplo:




```
// Empleado.java

import java.util.*;

public class Empleado {

        private String nombre;
        private Date fechaIncorporacion;

        public int calcularAntiguedad(){
                Date fechaActual = new Date();
                return fechaActual.getYear() - fechaIncorporacion.getYear();
        }
        public void setFechaIncorporacion(Date fechaIncorporacion) {
                this.fechaIncorporacion = fechaIncorporacion;
        }
        // Getter y setter
        public String getNombre() {
                return nombre;
        }
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
}
```

```
// Empresa.java

import java.util.*;

public class Empresa {

	private String razonSocial;
	private List<Empleado> empleados = new ArrayList<>(); // El atributo es una colección

	public void incorporarEmpleado(Empleado empleado){
		empleados.add(empleado);
	}
	public int cantidadEmpleados() {
		return empleados.size();
	}
	// Getters y setters
	public String getRazonSocial() {
		return razonSocial;
	}
	public void setRazonSocial(String razonSocial) {
		this.razonSocial = razonSocial;
	}
}
```

Igualdad y ordenamiento en las colecciones

Elementos iguales

En Java para determinar si dos elementos son iguales se deben sobreescribir los métodos equals() y hashCode().

Orden entre elementos

En Java para comparar objetos en las colecciones se debe implementar la interface Comparable. En el caso de elementos List se puede utilizar el método sort.

```
// Empleado.java
import java.util.*;

public class Empleado implements Comparable<Empleado>{

	// Otros atributos y métodos
	@Override
	public int compareTo(Empleado e2) {
		return this.getEdad() - e2.getEdad();
	}
}
```

```
// Empresa.java
import java.util.*;

public class Empresa {
	// Atributos, métodos

	public void mostrarEmpleadosOrdenadosPorEdad() {
		empleados.sort(null);
		mostrarEmpleados();
	}
}
```

Pendiente
Resolver ejercicio propuesto sobre colecciones

Sincrónico

Colecciones más utilizadas:
- ArrayList
- HashMap

Al implementar el método compareTo() el parámetro que se recibe es del tipo de la clase, ya que se especifica en la sentencia implements

```
public class Jugador implements Comparable<Jugador> {
	// código de la clase
}
```

Para ordenar los elementos de la lista:

plantel.sort(null); // El método requiere que se especifique un parámetro Comparator. Cuando se especifica el parámetro como null, se utiliza el compareTo como forma natural de comparar elementos

Cuando se imprime un objeto por consola, se ejecuta el método toString():

System.out.println(jug.toString());
System.out.println(jug); // Ambas opciones son equivalente

Cuando se requiere ordenar los elementos de varias formas diferentes, se requiere implementar Comparator. Sin emabrgo, estas funciones normalmente se delegan al DBMS.

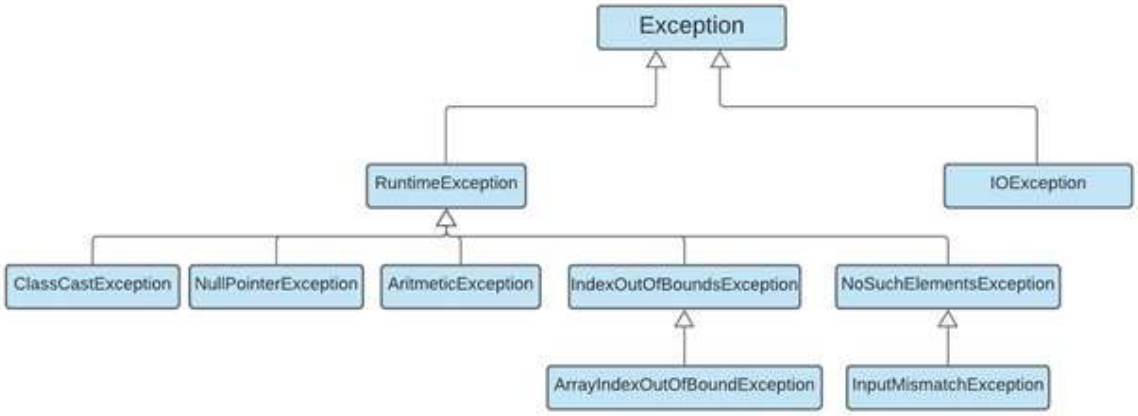
Sesión 17. Excepciones

Noviembre 24 de 2021

Ejemplo:

```
// En el bloque try se introduce el código que podría fallar
try {
        System.out.println("Primer número, debe ser un valor entero ");
        num1=scanner.nextInt();
        System.out.println(" Divisor, un valor entero ");
        num2=scanner.nextInt();
        division= num1/num2;
        System.out.println(division);
}
// En los bloques catch se incluyen las acciones que deberían realizarse en caso de atrapar un error del tipo especificado
catch(InputMismatchExcption excepcion) {
        System.err.println(“Se ingresó un tipo de dato incorrecto”);
}
catch(ArithmeticException excepcion) {
        System.err.println(“Se intentó dividir por cero”);
}
// Captura cualquier tipo de error no contemplado en los casos anteriores
catch(Exception e) {
        System.out.println("Error");
}
finally {
        // El bloque fnally es opcional y se ejecuta siempre
        System.out.println(“Ha finalizado el ejemplo”);
}
```

Algunos tipos de Excepciones



.

Lanzar una excepción

```
class Fecha {
        private int day;
        private int month;
        private int year;
        // Se especifica que el método puede generar una excepción. Es un método throweable.
        Al hacerlo, se obliga a que el código que llama al método esté protegido por un bloque
        try/catch
        public Fecha(int d, int m, int y) throws Exception {
                if (d<1||d>31||m<1||m>12)
                        throw new Exception("Los valores no son válidos");
                day=d;
                month=m;
                year=y;
        }
}

public static void main(String[] args) {
        try {
                Fecha fecha= new Fecha(100,-100,1000);
        catch (Exception e) {
                System.err.println(“No son valores válidos para una fecha”);
        }
}
```

Excepciones personalizadas

```
// FechaException.java
// La clase FechaException extiende a Exception
public class FechaException extends Exception{

        public FechaException() {
                super();
        }
        // Se crea un constructor con un parámetro para un mensaje personalizado
        public FechaException(String mensaje) {
                super(mensaje);
        }
        public String toString() {
                return "Se produjo la siguiente Excepción "+ this.getClass().getName() +"\n" +
                " Mensaje: " + this.getMessage() + "\n" ;
        }

}
```

```
// Fecha.java
class Fecha {

	private int day;
	private int month;
	private int year;

	public Fecha(int d, int m, int y) throws FechaException {
		if (d<1||d>31)
			// Se invoca el constructor de FechaException con un parámetro
			throw new FechaException("Error en el día");
		day=d;
		if (m<1||m>12)
			// Se invoca el constructor de FechaException con un parámetro
			throw new FechaException("Error en el mes");
		month=m;
		year=y;
	}
}

//main.java
public static void main(String[] args) {
	try{
		Fecha fecha= new Fecha(-1,10,2000);
	}
	catch(FechaException excepcion) {
		System.err.println(exception.getMessage());
		// Para que aparezca el mensaje personalizado completo se debería usar el
		// método toString()
		// System.err.println(exception.toString());
	}
}
```

Sincrónico:

```
System.err.println(e.toString()); // Imprime en la consola con color rojo

e.printStackTrace(); // Muestra la traza del error
```

```
Se produjo la siguiente excepcion: Error. No puede pasar su limite establecido.
    at presencial.Cliente.comprar(Cliente.java:20)
    at presencial.Main.main(Main.java:9)
```

Pregunta:
Si no se desea que el programa se corte, ¿cómo se debería manejar?

Sesión 18.

Sincrónico

Date y otros métodos para manejo de fechas

Los métodos after u before permiten comparar fechas:

```
Date fecha1 = new Date(121, 11, 25);
Date fecha2 = new Date(121, 11, 26);

if (fecha1.after(fecha2)) {
        System.out.println(fecha1 “ + está después de “ + fecha2);
} else if (fecha1.after(fecha2)){
        System.out.println(fecha1 “ + está antes de “ + fecha2);
} else {
        System.out.println(fecha1 “ es igual a “ + fecha2);
}
```

Hoy en día se recomienda utilizar:
LocalDate, LocalDateTime
calendar