Técnicas de Programación Avanzada (Java)

Curso 2021-2021

S_2 (Otros conceptos fundamentales de la Programación Orientada a Objetos) {

- 2.1 Herencia. Interfaces y clases abstractas. Agregación;
- 2.2 Polimorfismo;
- 2.3 Gestión de Excepciones;
- 2.4 Genericidad y plantillas;
- 2.5 Utilidades. Entrada y Salida;
- 2.6 Anotaciones;

}

Dra. Nieves Cubo Mateo

20 Sept y 04 Oct 2021

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

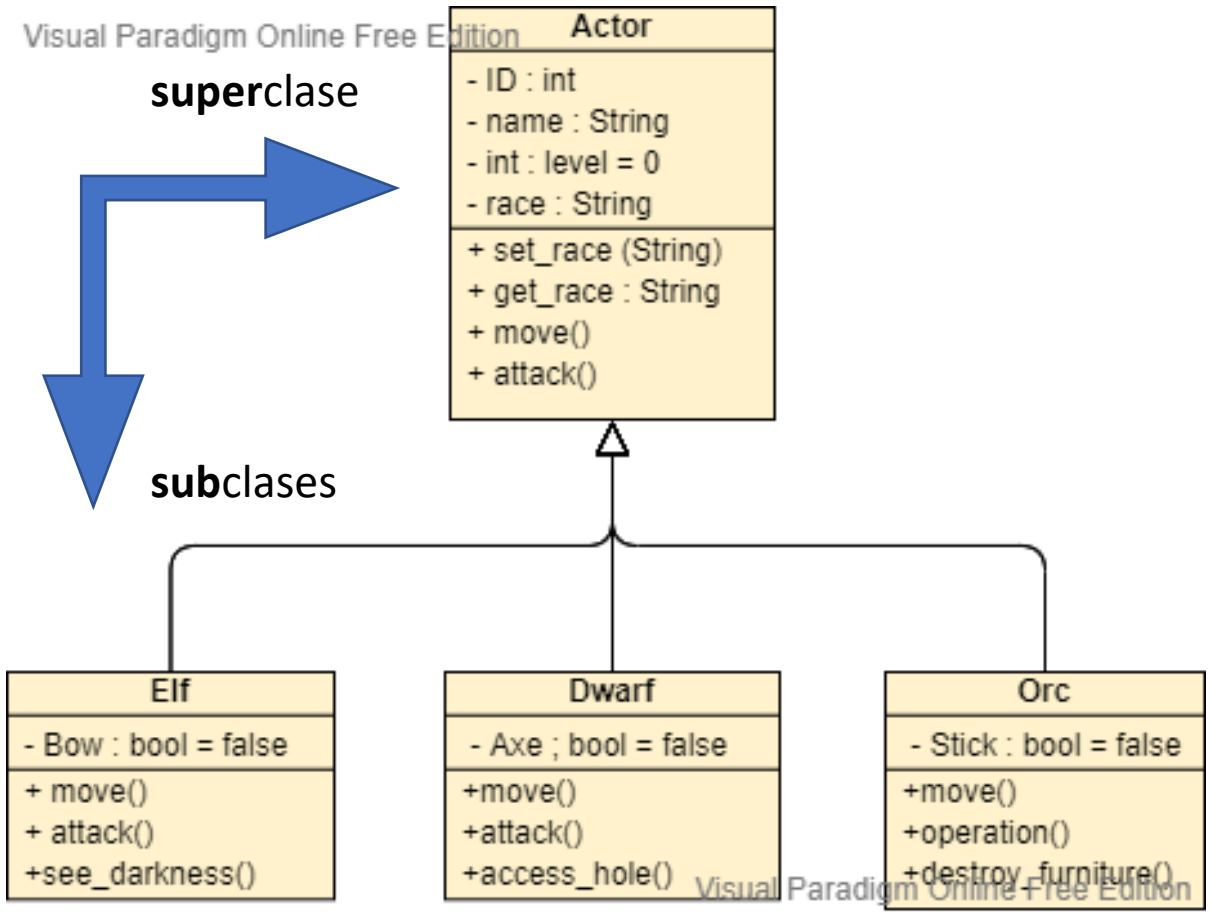
5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Herencia ("is a")



- Relación de extension. "Es un.."
- Se heredan los métodos y atributos privados o protegidos
- Los miembros privados no son accesibles por las clases hijas
- Las clases hijas pueden añadir sus propios atributos y métodos
- Sólo se representa la relación de clases (no objetos, ni número)
- Sólo se puede heredar de 1 clase
- Permite reutilizar código y agrupar objetos bajo una superclase pudiendo hacer uso del polimorfismo

class nombreSubclase **extends** nombreSuperclase{ }

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Herencia (“is a”)

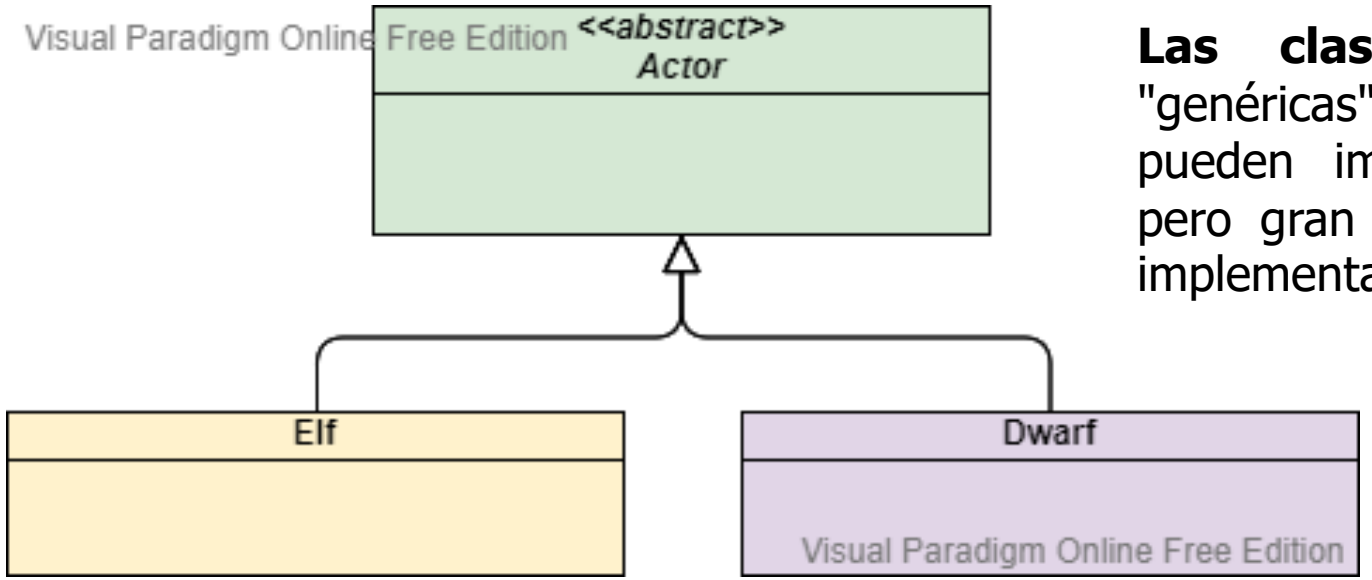
```
public class Actor {…}

public class Elf extends Actor{ }
public class Dwarf extends Actor{ }
```

```
ArrayList<Actor> Entities = new ArrayList<Actor>();

Entities.add(new Elf("Feanor", "Elf", “Explorer"));
Entities.add(new Dwarf("Durin", "Dwarf", “Warrior"));

for (Actor item:Entities) {
   item.talk();
}
```

```
Hi! My name is Feanor. I am an Elf.
Hi! My name is Durin. I am a Dwarf.
```

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Clases abstractas. “Incompletas” – ‘Qué’, pero no ‘cómo’



Las clases abstractas definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente, la conducta pero gran parte de la clase no está definida ni implementada.

No se hacen instancias de la clase abstracta, aunque si pueden hacerse arrays, listas, etc para albergar objetos de las clases hijas

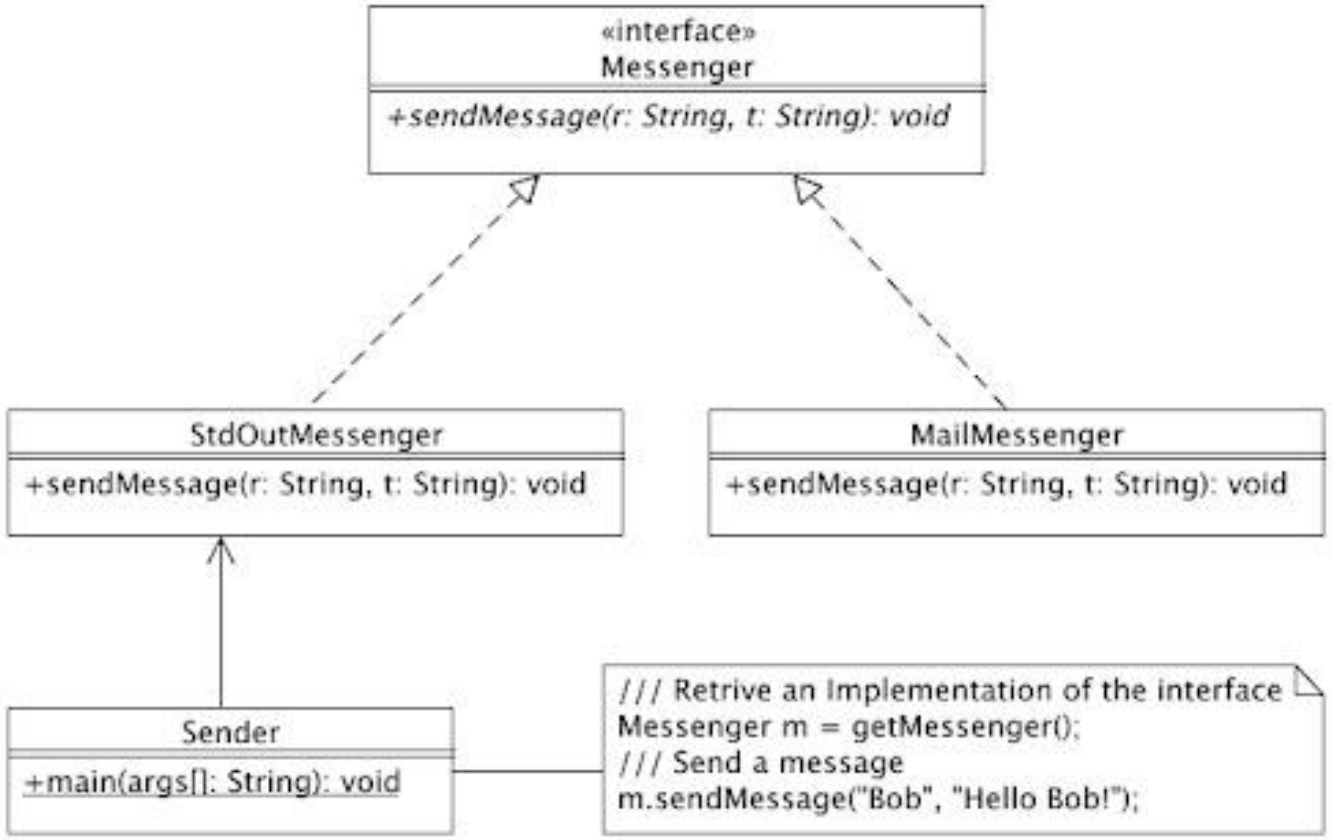
```
public abstract class Actor { }

public class Elf extends Actor{ }
```

```
Actor actor = new Actor(); //ERROR

Actor [] actores = new Actor[3];
actores[0] = new Elf();
```

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Interfaces – 'Qué', pero no 'cómo'

- Las interfaces son clases puramente abstractas
- No tienen atributos (new: puede tener constantes: static final)
- Poseen métodos sin implementar
- Para usarlas hay que implementarlas y definir todos sus métodos (si no se implementan todos, debe ser una clase abstracta)
- Se pueden implementar varias interfaces (separando nombres con comas)
- Las interfaces SI pueden heredar de varias interfaces. E implementar varias interfaces



class nombreSubclase **implements** nombreInterfaz{ }

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Interfaces vs clases abstractas

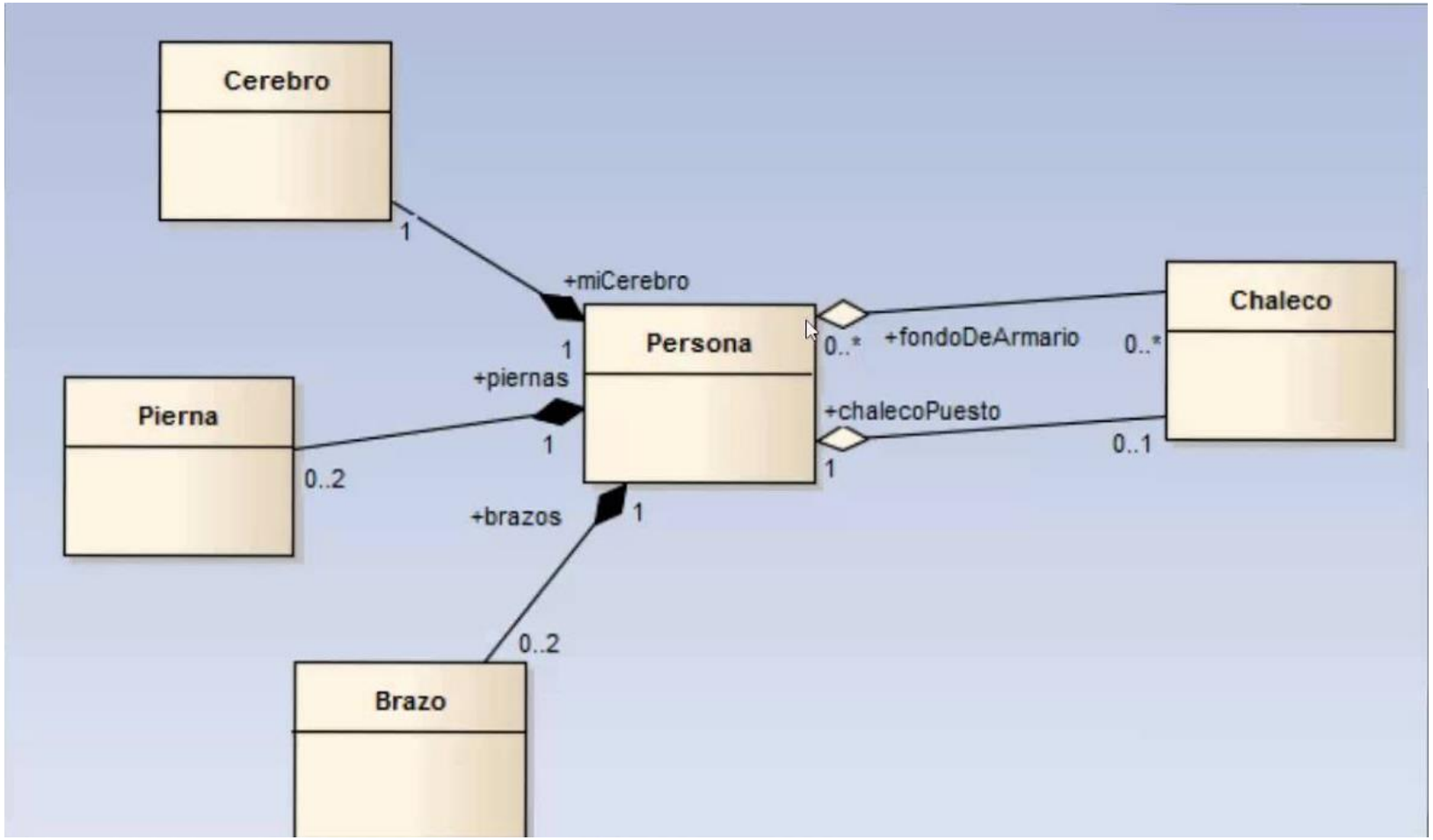
Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

Aunque no se instancie una clase abstracta, las clases hijas pueden llamar a su constructor en sus propios constructores.

Útil para inicializar valores.

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Agregación y Composición (“has a”)



Persona
- miCerebro:Cerebro - piernas[ ]: Pierna - brazos[ ]: Brazo
+ abrigar(chal:Chaleco);

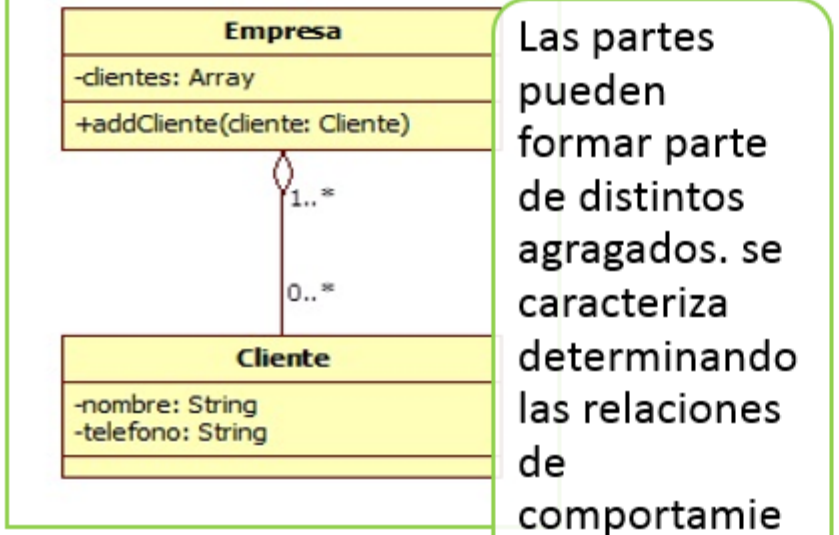
◇ **Agregación**
El chaleco puede existir sin la persona, y después de que esta desaparezca.

◆ **Composición**
Sin embargo, el cerebro, brazos y piernas…

2.1 Herencia. Interfaces y clases abstractas. Agregación.

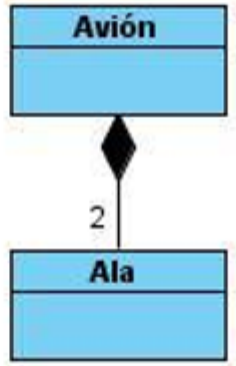
Agregación vs Composición (“has a”)

Agregación



Las partes pueden formar parte de distintos agragados. se caracteriza determinando las relaciones de comportamie nto y estructura.

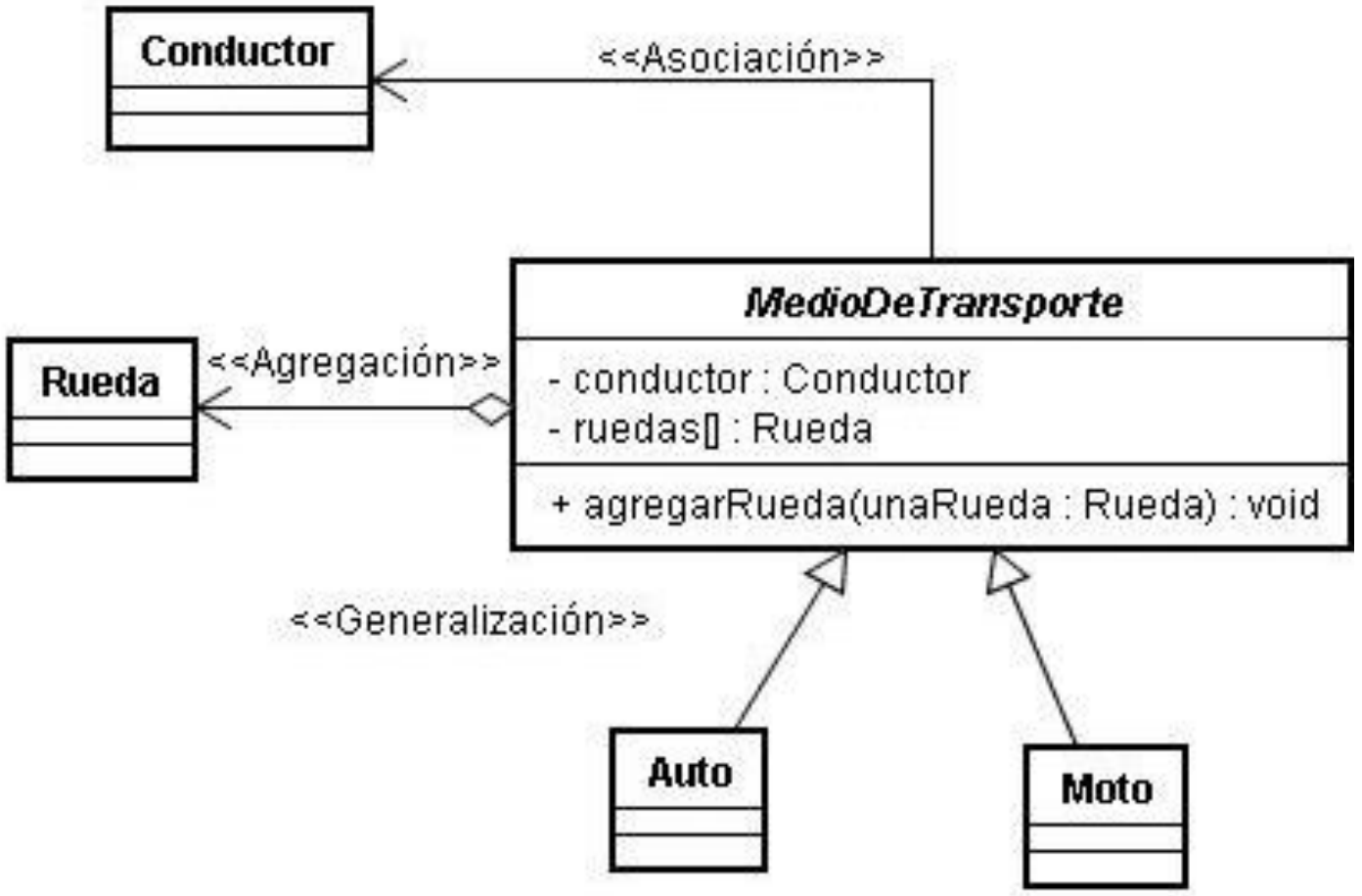
Compocisión



Agregción disyunta y estricta: las partes solo existen asociadas al compuesto (solo se accede a elllas a traves del compuesto

2.1 Herencia. Interfaces y clases abstractas. Agregación.

Resumen



<<Herencia>>

2.2 Polimorfismo

Uso

```
class Animal {
  public void makeSound() {
    System.out.println("Grr...");
  }
}
class Cat extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Meow");
  }
}
class Dog extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Woof");
  }
}
```

Recordad: En la sobrecarga, ambos métodos siguen vigentes. Aquí el método de la clase hija anula a la clase padre, se sobreescribe.

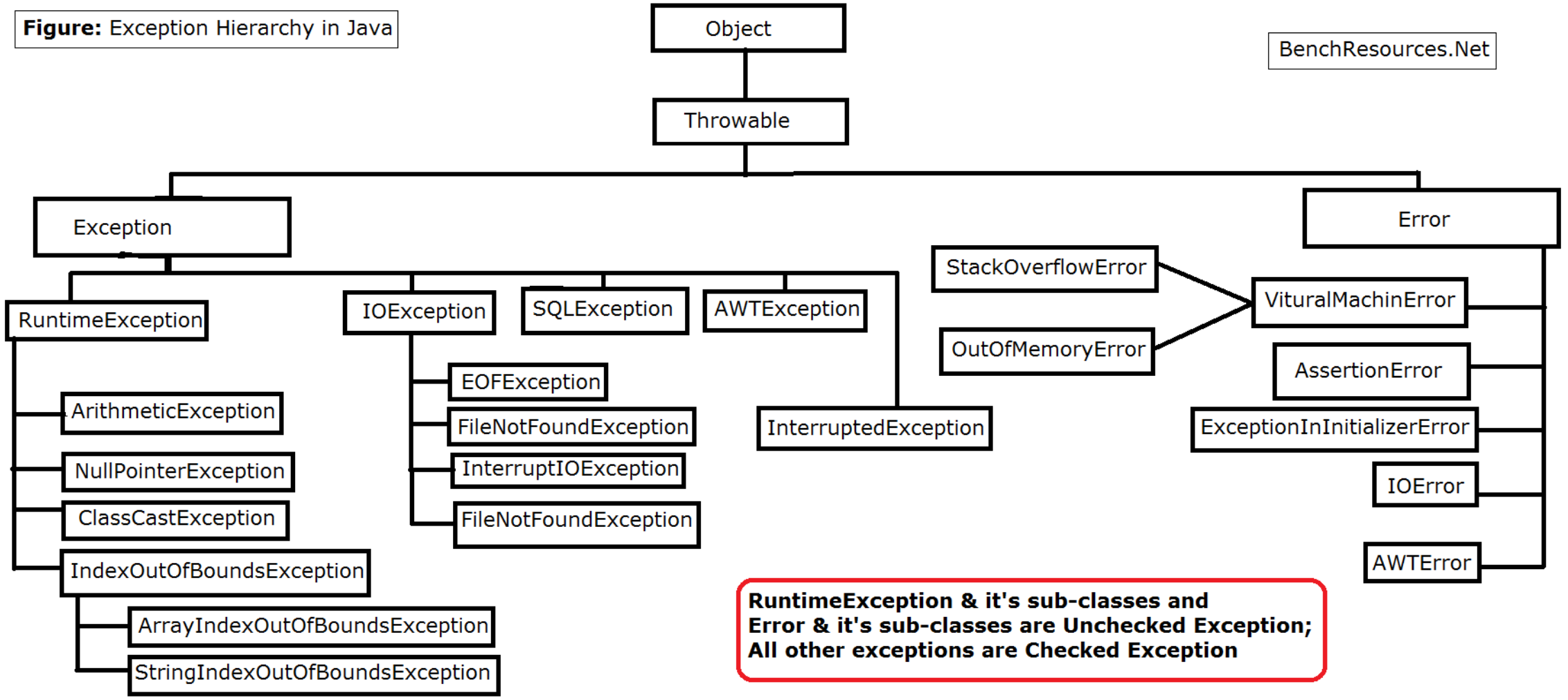
```
public class ClaseMain {

   public static void main(String[] args)
   {

      Animal [] animales = new Animal[2];

      animales[0] = new Dog();
      animales[1] = new Cat();

      for (Animal item:animales) {
         item.makeSound();
      }
   }
}
```

Polimorfismo, muchas formas. Allí donde se pueda usar el objeto padre, también se podrá hacer usa de cualquiera de sus hijas.

Principio de sustitución de Liskov (SOLID)

2.3 Gestión de Excepciones

En Java las excepciones son objetos “lanzables”



2.3 Gestión de Excepciones

Excepción vs Error

<table>
<tr><th colspan="2">Excepción</th><th>Error</th></tr>
<tr><td colspan="2">Situaciones que deberían solucionarse. El programa debería incluir bloques de try and catch para gestionarlos y recuperarse.</td><td rowspan="2">Problema grave debido a falta de recursos del sistema (run out memory, JVM error).</td></tr>
<tr><th>RuntimeException</th><th>IOException</th></tr>
<tr><td>Errores del programador (division por cero, acceso fuera de los límites de un array..). Gestión opcional</td><td>Errores que no puede evitar el programados, relacionados con E/S del programa. Gestión obligada.</td><td>El programa no debería “coger” estos errores. No es recuperable. Terminación del programa.</td></tr>
<tr><td>ArrayIndexOutOfBoundsException, NullPointerException</td><td></td><td>OutOfMemoryError ,IOError</td></tr>
</table>

2.3 Gestión de Excepciones

En Java las excepciones son objetos “lanzables”

Todos los errores y excepciones son subclases de Throwable, por lo que podrán acceder a sus métodos[1]. Los métodos más utilizados son los siguientes:

- **getMessage( )** Se usa para obtener un mensaje de error asociado con una excepción.

- printStackTrace(PrintStream s) Se utiliza para imprimir el registro del stack[2] donde se ha iniciado la excepción. Para recoger la info también se puede usar **getStackTrace ().**

- toString( ) Se utiliza para mostrar el nombre de una excepción junto con el mensaje que devuelve getMessage().

1 https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html

2 https://www.scalyr.com/blog/java-stack-trace-understanding/

2.3 Gestión de Excepciones

Excepción del sistema

```java
public class ClaseMain {
   public static void main(String[] args)
   {
      System.out.print(13/0);
   }

}
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at ejemplos.ClaseMain.main(ClaseMain.java:9)
```

2.3 Gestión de Excepciones

Uso de los bloques *try* y *catch [&finally]*

```java
package ejemplos;


public class ClaseMain {
  public static void main(String[] args)
  {
     try{
        System.out.println(13/0);
     }
     catch(ArithmeticException myExcep) {
        System.err.println("Error artimético");
     }
     finally{
        System.out.println("FIN");
     }
   }
}
```

```
Error artimético
FIN
```

2.3 Gestión de Excepciones

Ejemplo: ejercicio 2

```java
public static void main(String[] args) {
  Scanner scan = new Scanner(System.in);
  int n=0;
  boolean cont=true;

  do{
    try {
      System.out.print("\nIntroduzca un número entero:");
      n = scan.nextInt();
      cont=false;

    } catch (InputMismatchException e) {
      System.out.println("Debe introducir un número entero.");
      scan.nextLine();
    }
  }while(cont);

  System.out.println((esPrimo(n))?"Es primo": "No es primo");
}
```

```java
boolean esPrimo(int n) {
  for(int i=2;i<n;i++) {
    if(n%i==0) {
      return false;
    }
  }
  return true;
}
```

Ejercicio 2

2.3 Gestión de Excepciones

Ejemplo: ejercicio 2

```
Scanner scan = new Scanner(System.in);

double suma = 0;
boolean cont = true;

ArrayList<Double> array = new ArrayList<Double>();

System.out.print("Introduzca números hasta que lo desee. Cuando quiera terminar introduzca 'x':\n");

do{
  try {

  array.add(scan.nextDouble());

  } catch (InputMismatchException e) {
     if (scan.nextLine().equals("x")){
       for (double item:array) {
          suma += item;
       }
      System.out.println("La suma de todos los valores introducidos es: " + suma);
     }
     else {System.out.println("Debe introducir un número.");}
  }
}while(cont);
}
```

```
for (int i = 0; i < array.size(); i ++) {
   suma += array.get(i);}
```

Ejercicio 2

```
if (scan.nextLine().equals("x")){
  System.out.println("Los valores introducidos son: " );
  for (double item: array) {
    System.out.print(item + ", ");}
}
```

Ejercicio 1

Ejercicio 4

2.3 Gestión de Excepciones

Creación de excepciones propias

```
public class ValorNoValidoException extends Exception {
    public ValorNoValidoException() {
        super("El valor introducido no es válido");
    }
}
```

```
public class Checker{
    public double checkValue(double valor) throws ValorNoValidoException {
        if(valor < 0) throw new ValorNoValidoException ();
        return valor;
    }
}
```

```
Checker checker = new Checker();
try {
    checker.checkValue(-20);
} catch (ValorNoValidoException e) {
    e.printStackTrace();
}
```

2.4 Genericidad y Plantillas

Generics Java (Templates C++) - Métodos

```java
public void printArray(double [] array) {
    for (double item:array) {
        System.out.println(item);
    }
}
```

```java
public void printArray(int [] array) {
    for (int item:array) {
        System.out.println(item);
    }
}
```

Sobrecarga de métodos para aceptar diefrentes tipos de datos.

Repetición de Código…
Poco eficiente….

GENERICS ↓

```java
public <T> void printArray(T[] array) {
    for (T item:array) {
        System.out.println(item);
    }
}
```

2.4 Genericidad y Plantillas

Generics Java (Templates C++) - Clases

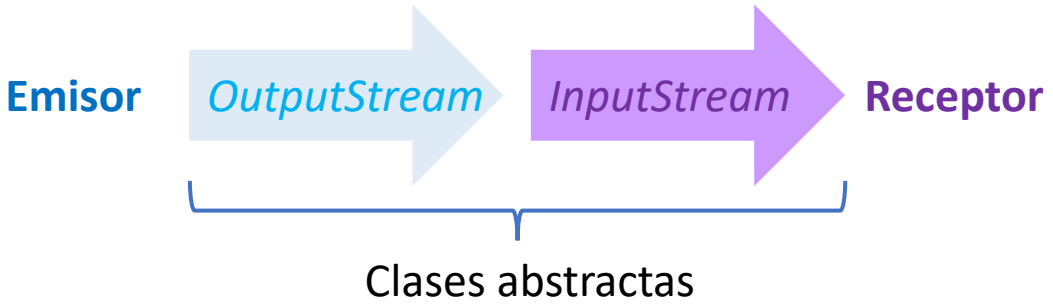
```
static class Pair<T> {
    private T left, right;
    public Pair(T left, T right) {
    this.left = left;
    this.right = right;
    }
    public T getLeft() {return left;}
    public T getRight() {return right;}
}
```

```
public static void main (String[] args)  {
    Pair<Integer> i = new Pair(1,2);
    Pair<String> s = new Pair("hello", "world");
    Pair<Float> f = new Pair(1,2);

    System.out.println(i.getLeft());
    System.out.println(s.getRight());
    System.out.println(f.getRight());
}
```

2.5 Utilidades entrada y salida

Standard, archivos y streams

java.io → clases e interfaces para la gestión de **flujos de datos (*streams*)** y administración de buffers



En Java se accede a la E/S estándar a través de campos públicos y estáticos de la clase *java.lang.System*:

- **System.in [InputStream]:** implementa la entrada estándar (← *Scanner* espera recibir este stream)
- **System.out [PrintStream]:** implementa la salida estándar
- **System.err** implementa la salida de error estándar

- El sistema abre y cierra los flujos standard de forma automática
- Cualquier fallo durante el proceso lanza la excepción **IOException**

2.5 Utilidades entrada y salida

Cambiar entrada estándar

Cambio de entrada standard: *System.setIn*

Ejemplo: **cambiamos la entrada por defecto a un archivo** ➔

Importante: uso de gestión de excepciones con *try y catch*, también en el *finally*

De igual manera puede modificarse la salida estándar de error (*System.err()).*

```
FileInputStream fis = null;
Scanner scanner = new Scanner(System.in);

try {
  fis = new FileInputStream("entrada.txt");
  System.setIn(fis);

  while(scanner.hasNext()) {
    String s = scanner.next();
    System.out.println(s);
  }
}catch (FileNotFoundException e) {
  e.printStackTrace();
}finally {
  scanner.close();
  if (fis!=null) {
    try {
      fis.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

2.5 Utilidades entrada y salida

Cambiar salida estándar

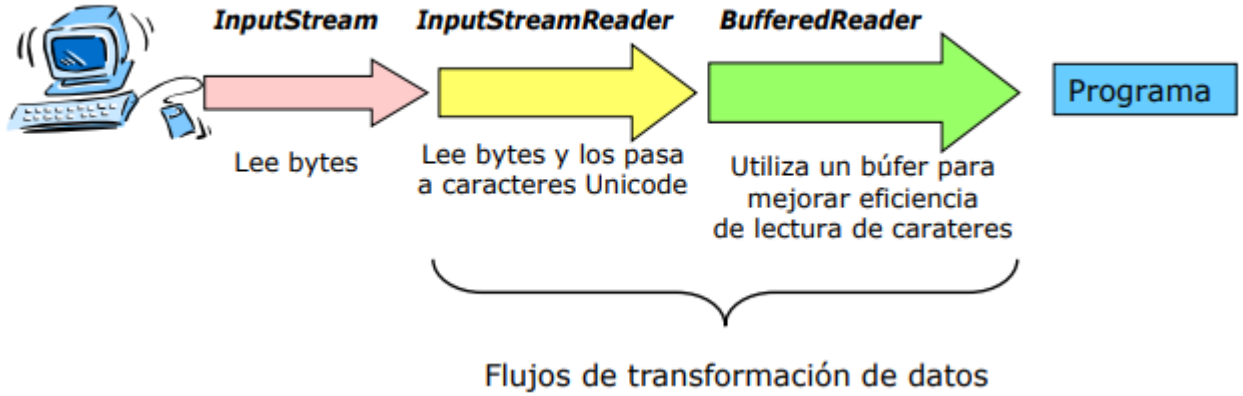
Cambio de salida standard: *System.setOut*

Ejemplo: **cambiamos la salida por defecto a un archivo**:

```
import java.io.FileOutputStream;

FileOutputStream fos = new FileOutputStream("salida.txt");
System.setOut(new PrintStream(fos));

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

fos.close();
```

2.5 Utilidades entrada y salida

Archivos



- **Archivos de texto:**
  FileReader, FileWriter → readLine(), println()
- **Archivos binarios:**
  InputStream, Outpustream → read(), write()

Si usamos sólo **FileInputStream**, **FileOuputStream**, **FileReader** o **FileWriter**, cada vez que hagamos una lectura o escritura, se ***hará fisicamente en el disco duro***. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro.

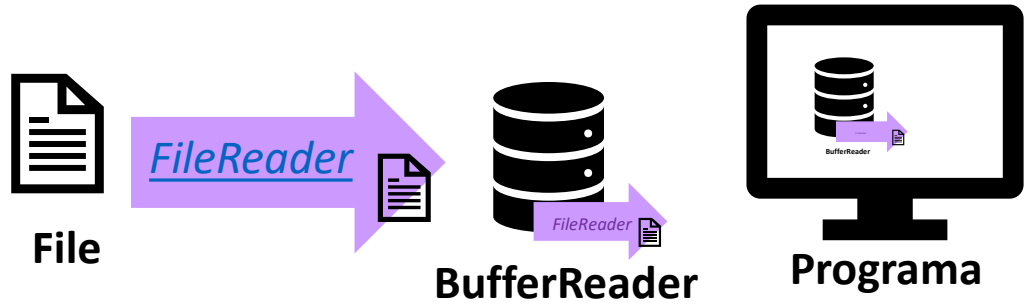
Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.
• Si vamos escribiendo, se guardará los datos hasta que tenga bastantes datos como para hacer la escritura eficiente.
• Si queremos leer, la clase leerá muchos datos de golpe, aunque sólo nos dé los que hayamos pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez.

Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido. La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.

2.5 Utilidades entrada y salida

Archivos, lectura – java.io.Reader

```
File archivo = null;
FileReader fr = null;
BufferedReader br = null;

try {
   archivo = new File ("C:\\archivo.txt");
   fr = new FileReader (archivo);
   br = new BufferedReader(fr);

   String linea;
   while((linea=br.readLine())!=null)
      System.out.println(linea);
}
catch(FileNotFoundException e){
   e.printStackTrace();
}finally{
   try{
      if( null != fr ){
         fr.close();
      }
   }catch (IOException e2){
      e2.printStackTrace();
   }
}
```



2.5 Utilidades entrada y salida

Archivos, escritura – java.io.Writer

```
FileWriter fichero = null;
PrintWriter pw = null;
try
{
    fichero = new FileWriter("c:/Prueba.txt");
    pw = new PrintWriter(fichero);

    for (int i = 0; i < 10; i++)
        pw.println("Linea " + i);

} catch (Exception e) {
    e.printStackTrace();
} finally {
   try {
   if (null != fichero)
      fichero.close();
   } catch (Exception e2) {
      e2.printStackTrace();
   }
}
```

Si queremos añadir al final de un fichero ya existente, simplemente debemos poner un *flag* a true como segundo parámetro del constructor de *FileWriter*.

```
FileWriter fichero =
new FileWriter("c:/prueba.txt",true);
```

2.6 Anotaciones

"Post-it" informáticos

- No modifican la actividad de un programa ordenado → Pero si tienen efecto, pueden cambiar la forma en la que el compilador trata el programa. Ayudan a relacionar metadatos (info) con componentes del programa
- Si no cumplimos lo mencionado en las @notaciones, el programa puede dar warning o errores

- **De java.lang.annotation (normalmente usados en programación de anotaciones propias):**
  - ***@Retention:*** política de retención (tiempo y ámbito en el que una anotación está presente durante el proceso de compilación y despliegue [@Retention(RetentionPolicy.RUNTIME/CLASS/SOURCE)]).
  - ***@Documented:*** indica a una herramienta que se debe documentar una anotación (en nuestro caso que debe aparecer como clase cuando JavaDocs genera la documentación)
  - ***@Target:*** Especifica los tipos de elementos a los que se puede aplicar una anotación propia
  - ***@Inherited****:* hace que la anotación de una superclase sea heredada por una subclase.

- **De java.lang:**
  - ***@Override:*** para asegurar que un método de superclase esté anulado y no simplemente sobrecargado (si se borra el método de la clase padre nos daremos cuentas gracias a esto)
  - **@Deprecated**: informa al programa de que un método, clase o campo está obsoleto y no debería ser utilizado ya. Se puede añadir un JavaDocs con la nueva alternativa.
  - **@SafeVarargs:** el programador confirma que el método o constructor no hace ninguna operación potencialmente insegura en su parámetros *varargs*
  - **@SuppressWarnings:** suprime los warnings generados en un método

2.6 Anotaciones

“Post-it” informáticos – Creación de anotaciones propias

- **@interface** *NombreAnotacion*
- Contiene campos en su interior de tipo primitive (arrays también)
- @Target puede acompanar para definir a qué elementos se puede aplicar (anotaciones, constructores, campos, variables locales, métodos, paquetes, parámetros, etc)

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
public @interface MyAnnotation {

    String   value();
    String[] names();
    int      age();
}
```

*JavaDocs

Documentando el código

- Genera documentación automática para nuestro programa si hacemos uso de los tags
- Podemos acompañar a las anotaciones de su correspondiente JavaDocs para dar más información

```
@Deprecated
/**
  @deprecated Use MyNewComponent instead.
*/
public class MyComponent {

}
```

- Tutorial official de JavaDocs: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html
- Más tutorials:
  - https://www.baeldung.com/javadoc
  - Cómo generar Javadoc en Eclipse: https://www.tutorialspoint.com/How-to-write-generate-and-use-Javadoc-in-Eclipse

*JavaDocs

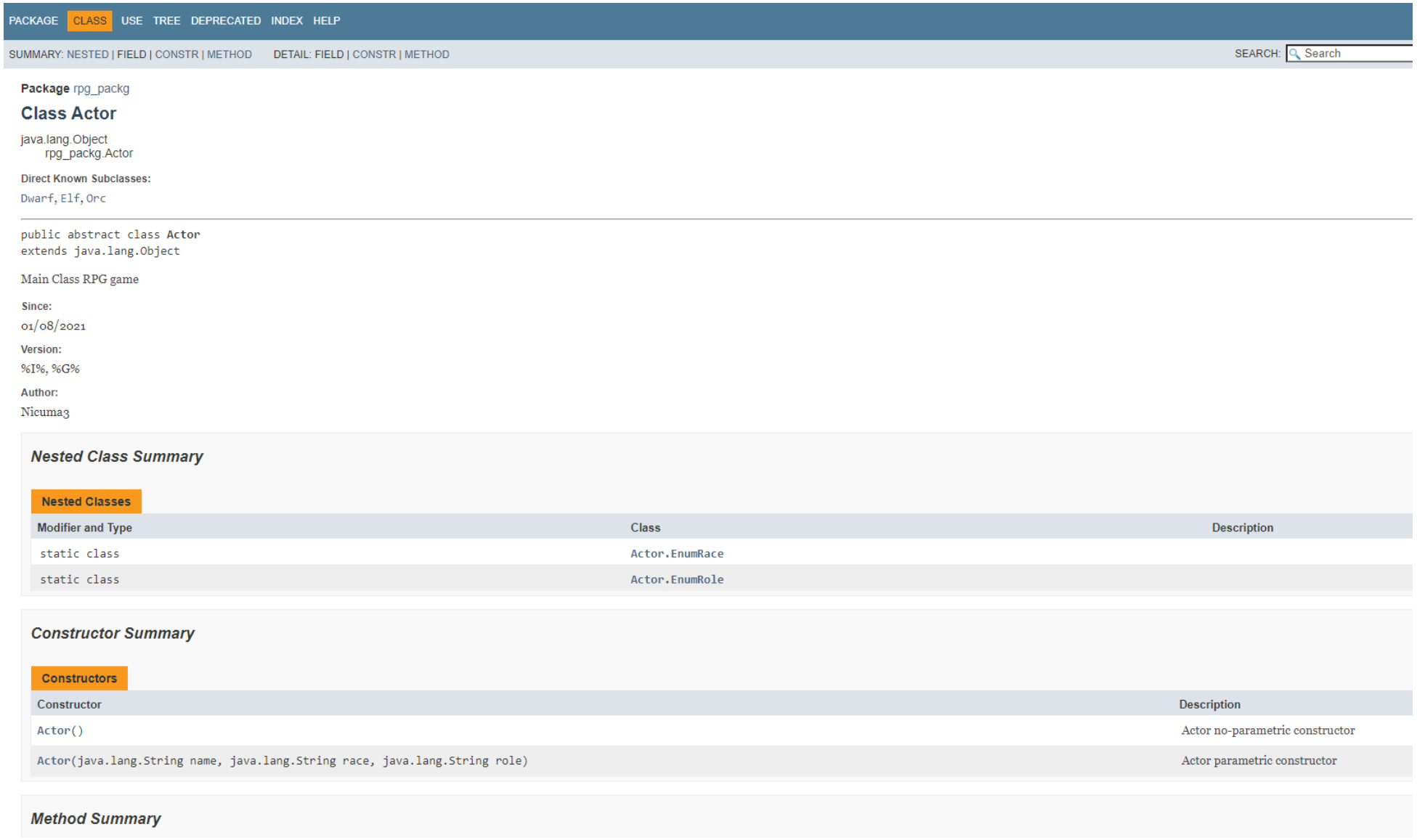
Ejemplo

```java
package rpg_packg;

/**
 *  Main Class RPG game
 *
 *  @author Nicuma3
 *  @version %I%, %G%
 *  @since 01/08/2021
 */
public abstract class Actor {


//Member attributes

/**
 * Character ID.
 */
int ID;

/**
 * Character Race. Elf, Dwarf, Orc, ...
 */
private String race;
//private Races ER;

/**
 * Character Role. Wizard, Warrior, Explorer...
 */
private String role;
```

```java
//Member methods
/**
 * Actor no-parametric constructor
 */
public Actor() {

}

/**
 * Actor parametric constructor
 * @param name Actor's name
 * @param race Actor's race (Elf, dwarf, orc, [...])
 * @param role Actor's role (Wizard, explorer, warrior,
[...])
 */
public Actor(String name, String race, String role) {
this.name = name;
this.race = race;
this.role = role;
}
```

*JavaDocs

Ejemplo



PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH: Search

Package rpg_packg

Class Actor

java.lang.Object
    rpg_packg.Actor

Direct Known Subclasses:

Dwarf, Elf, Orc

```
public abstract class Actor
extends java.lang.Object
```

Main Class RPG game

Since:

01/08/2021

Version:

%I%, %G%

Author:

Nicuma3

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description
static class	Actor.EnumRace	
static class	Actor.EnumRole	

Constructor Summary

Constructors

Constructor	Description
Actor()	Actor no-parametric constructor
Actor(java.lang.String name, java.lang.String race, java.lang.String role)	Actor parametric constructor

Method Summary

- DUDAS mails

- REPASO CONCEPTOS IMPORTANTES

- REPASO ERRORES RPGs

Temas a tratar

Dudas recibidas a lo largo de la semana o al corregir los UML

- **Enums:** qué son, como usar, valores vs atributos, sacar array con los posibles valores
- **Interfaces:** ejemplo de uso
- **Herencia** realmente necesaria?
- No confundir herencia con **diferentes objetos** de una misma clase, pero distintos atributos
- **No hago varias clases iguales**. Después hago varios objetos de una clase si es necesario, pasando atributos
- No repetir método en todas las clases hijas si no es diferente la implementación (polimorfismo)
- **Modificadores:** static, final, abstract

REVISAD LAS CORRECCIONES DE LOS UML!!
(en general de cualquier cosa que entreguéis)

*Enums

Enumeraciones: qué son, como usar, valores vs atributos, sacar array con los posibles valores

```
public enum EnumRace{
  ELF, DWARF, ORC;
}
```

```
public enum EnumRace{
  ELF("Elf"), DWARF("Dwarf"), ORC("Orc");

  Private final String Race;

  EnumRace(String Race) {
    this.Race = Race;
  }

  public String getRace() {
    return Race;
  }
}
```

- Permite crear un set de valores posibles para una variable
- Son clases estáticas y constantes

*Enums

Enumeraciones: qué son, como usar, valores vs atributos, sacar array con los posibles valores

```
public enum EnumRace{
  ELF, DWARF, ORC;
}
```

- Cómo usamos sus valores y cómo accedemos a ellos desde if y switch

```
public String testEnum(EnumRace race) {
  if (race==EnumRace.ELF)
    return ("It's an elf!");
      else if (race==EnumRace.DWARF)
        return ("It's a dwarf!");
          else if (race==EnumRace.ORC)
            return ("It's an orc!");
          else
            return("No vslid type");
}
```

```
public String testEnum(EnumRace race) {
  switch(race) {
  case ELF:
    return ("It's an elf!");
  case DWARF:
    return ("It's a dwarf!");
  case ORC:
    return ("It's an orc!");
  default:
    return("No vslid type");
  }
}
```

*Enums

Enumeraciones: qué son, como usar, valores vs atributos, sacar array con los posibles valores

```java
public enum EnumRace{
  ELF("Elf"), DWARF("Dwarf"), ORC("Orc");

  Private final String Race;

  EnumRace(String Race) {
    this.Race = Race;
  }

  public String getRace() {
    return Race;
  }
}
```

```java
public static ArrayList<String> enumIteration() {
  EnumRace[] races = EnumRace.values();
  ArrayList<String> stringRace = new ArrayList<String>();
  for (EnumRace race : races) {
    stringRace.add(race.toString()); //race.getRace()
  }
  return stringRace;
}
```

*static, abstract, final

Modificadores

	Clases	Atributos/Variables	Métodos/Funciones	Otros
static	NO. Sólo con clases interiores (anidadas). Indica que se trata de un atributo estático	Variable compartida por todas las instancias de una clase de forma “global”.	Un método estático solo puede llamar a otros métodos estáticos. Pueden acceder a los datos de tipo estático directamente, sin necesidad de objetos.	Bloques: solo se ejecuta una vez, cuando la clase se inicializa por primera vez
final	No puede ser heredada (no puede ser abstracta, tiene que ser completa)	No puede ser variado su valor	No puede ser sobreescrito	-
abstract	No pueden ser instanciadas. Deben contener al menos un método abstracto, que debe ser definido en las clases hijas.	No	No está definido. Debe definirse en otras clases	-

*static, abstract, final

Modificadores

Modifiers-Elements Matrix in Java

element / modifier	Data field	Method	Constructor	Class		Interface	
				top level (outer)	nested (inner)	top level (outer)	nested (inner)
abstract	no	yes	no	yes	yes	yes	yes
final	yes	yes	no	yes	yes	no	no
native	no	yes	no	no	no	no	no
private	yes	yes	yes	no	yes	no	yes
protected	yes	yes	yes	no	yes	no	yes
public	yes	yes	yes	yes	yes	yes	yes
static	yes	yes	no	no	yes	no	yes
synchronized	no	yes	no	no	no	no	no

*Clases abstractas

Ejemplo de uso

- Una superclase permite unificar campos y métodos de las subclases, evitando la repetición de código y unificando procesos.
- Si **no se va a instanciar** (hacer objetos), la clase puede hacerse abstracta
- Si la clase es abstracta **debe tener al menos un método abstracto** (sólo declarado, no implementado), que será implementado en las subclases.

```java
public abstract class Actor {

public int ID;
private String race;
private String role;
private String name; //Nickname

public abstract void identify();

public void talk() {
  System.out.print("Hi! My name is " +
this.name + ". ");
}
```

```java
public class Elf extends Actor {//Inherits from Actor

@Override
public void talk() {
  super.talk();
  System.out.println("I am an Elf. ");
}

public void identify() {
   System.out.println("Elf");
}
```

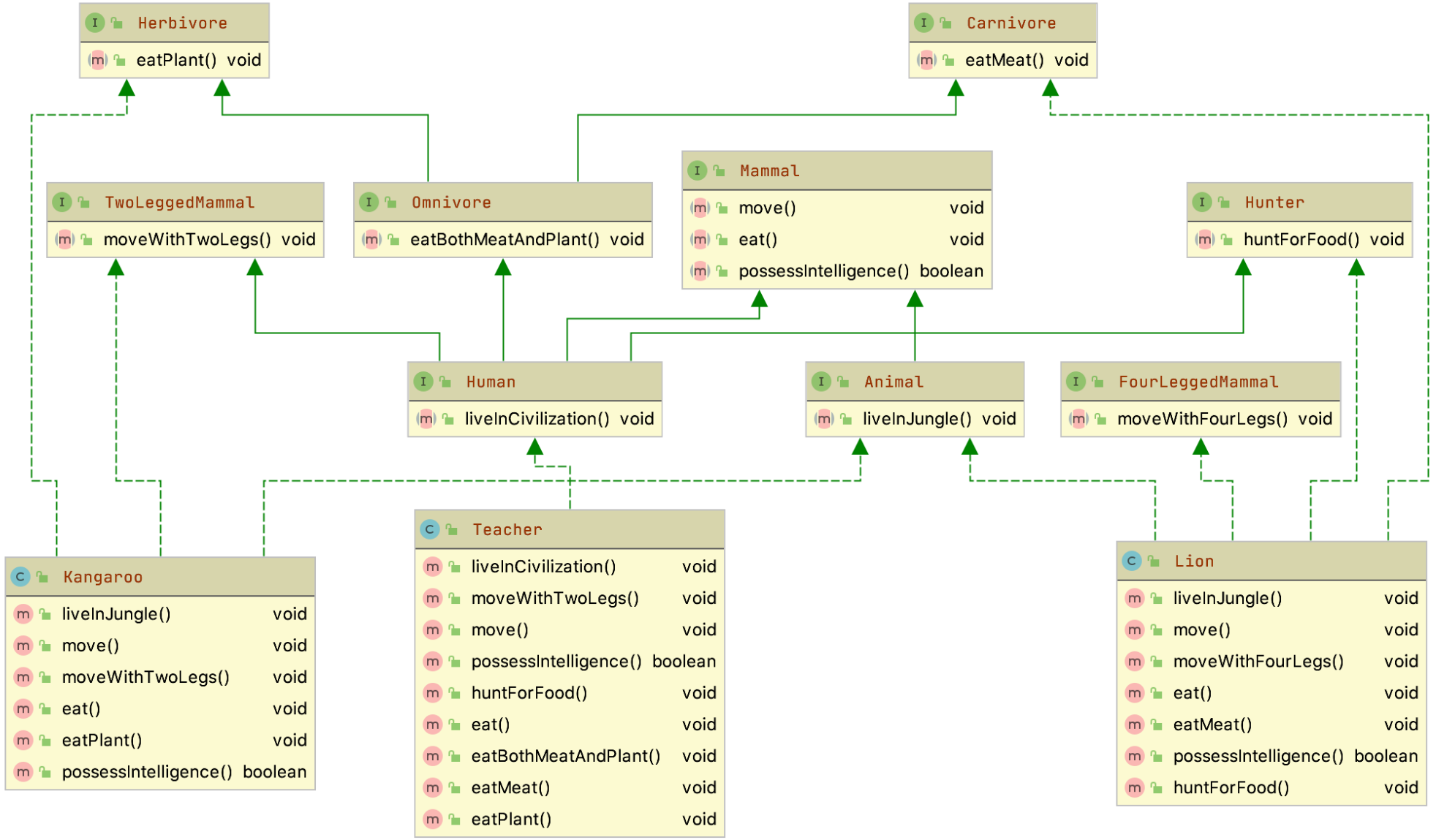
*Interfaces

Ejemplo de uso

- Son **clases abstractas puras** (todos los métodos son abstractos, no están definidos). *¿Qué? Pero no ¿cómo?*
- En principio, no tienen campos (atributos), sólo métodos
- **Una clase puede implementar más de una interfaz. Pero sólo heredar de una clase**
- Se usa para “forzar” que una clase que pertenece a diferentes “grupos” (interfaces), tenga que implementar obligatoriamente una serie de métodos necesarios para cada grupo.
- Si implementa dos interfaces con el mismo método y el mismo tipo de dato de retorno, no hay problema, ya que debe implementar que quiere que haga y no habrá ambigüedad de uso.
- Pero, Podemos generar un error si implementamos dos interfaces con el mismo método pero diferentes tipos de datos de retorno.

*Interfaces

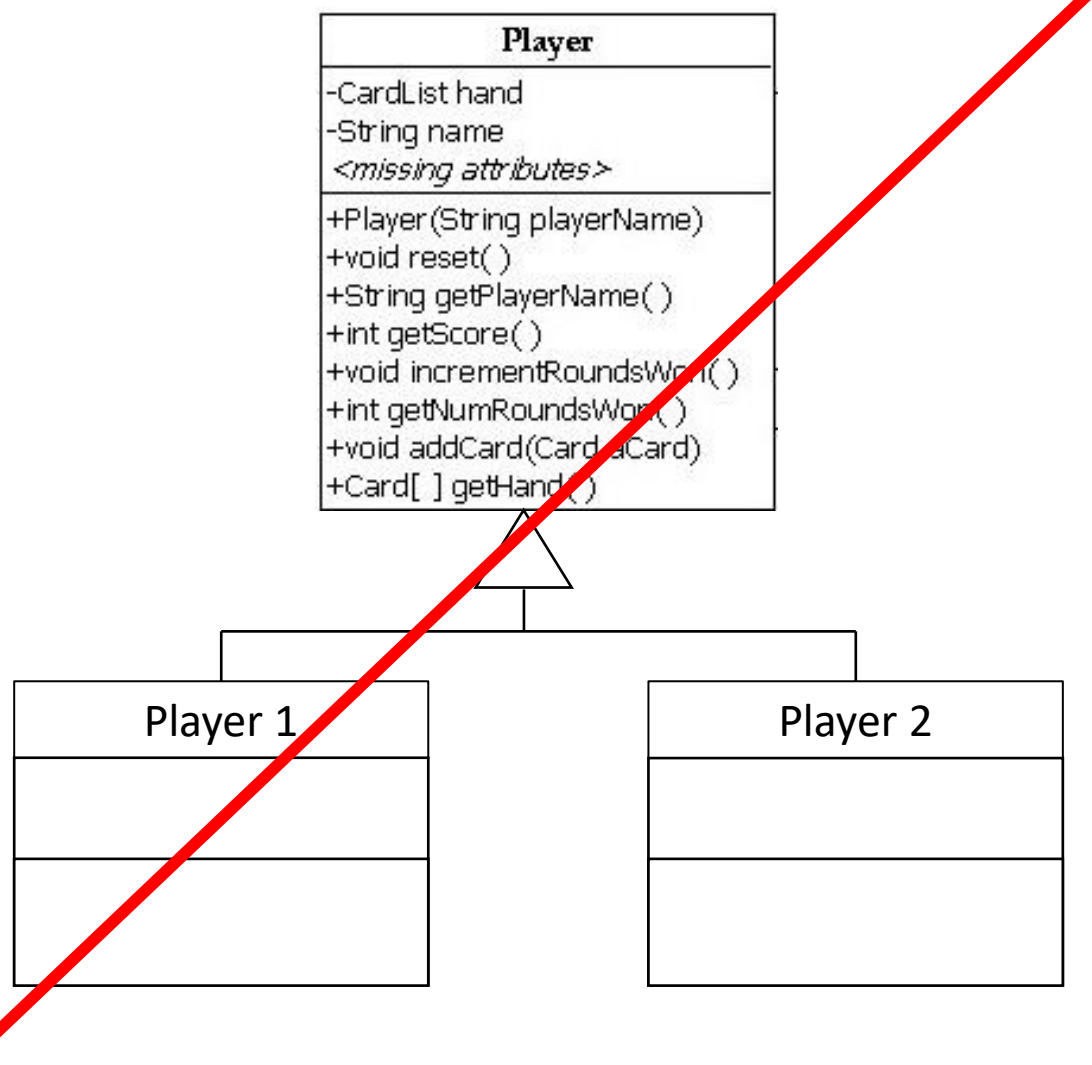
Ejemplo de uso



https://stackoverflow.com/questions/21263607/can-a-normal-class-implement-multiple-interfaces

*Clase vs Objeto.

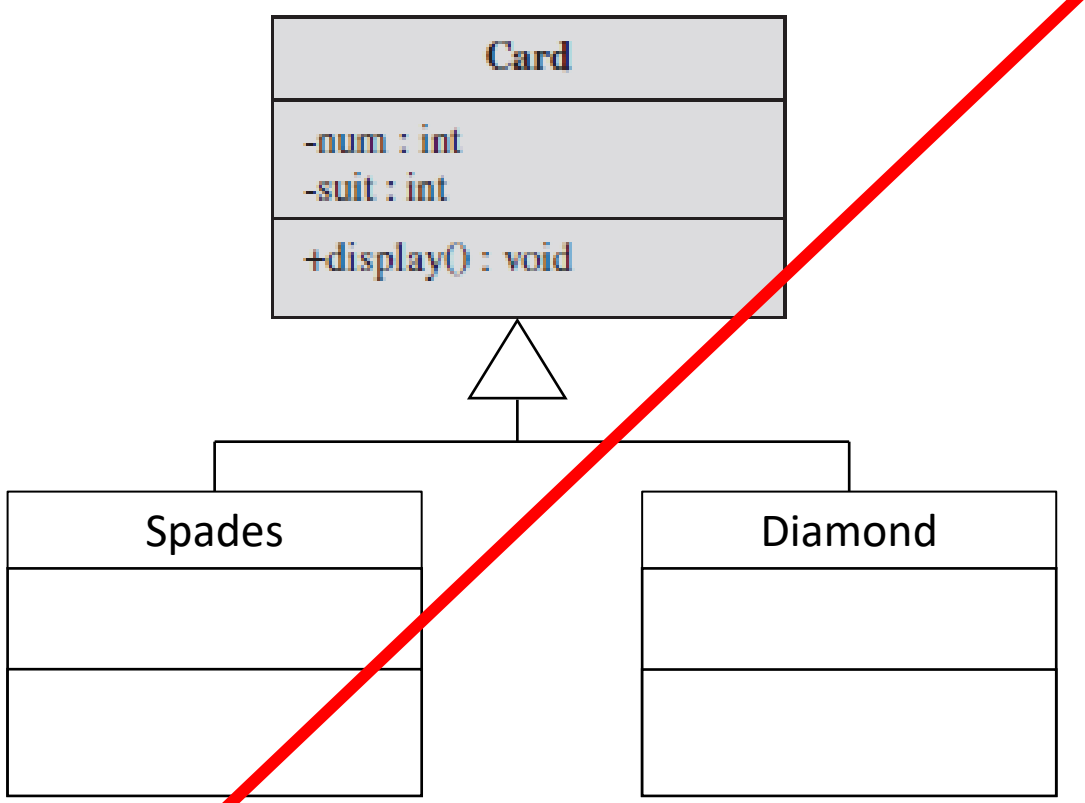
Cuidado!




```java
public class Player { … }

public class MainClass {

public static void main(String[] args) {

Player P1 = new Player(args);
Player P2 = new Player(args_dif);
}
```

*Herencia vs Clase(diferentes parámetros)

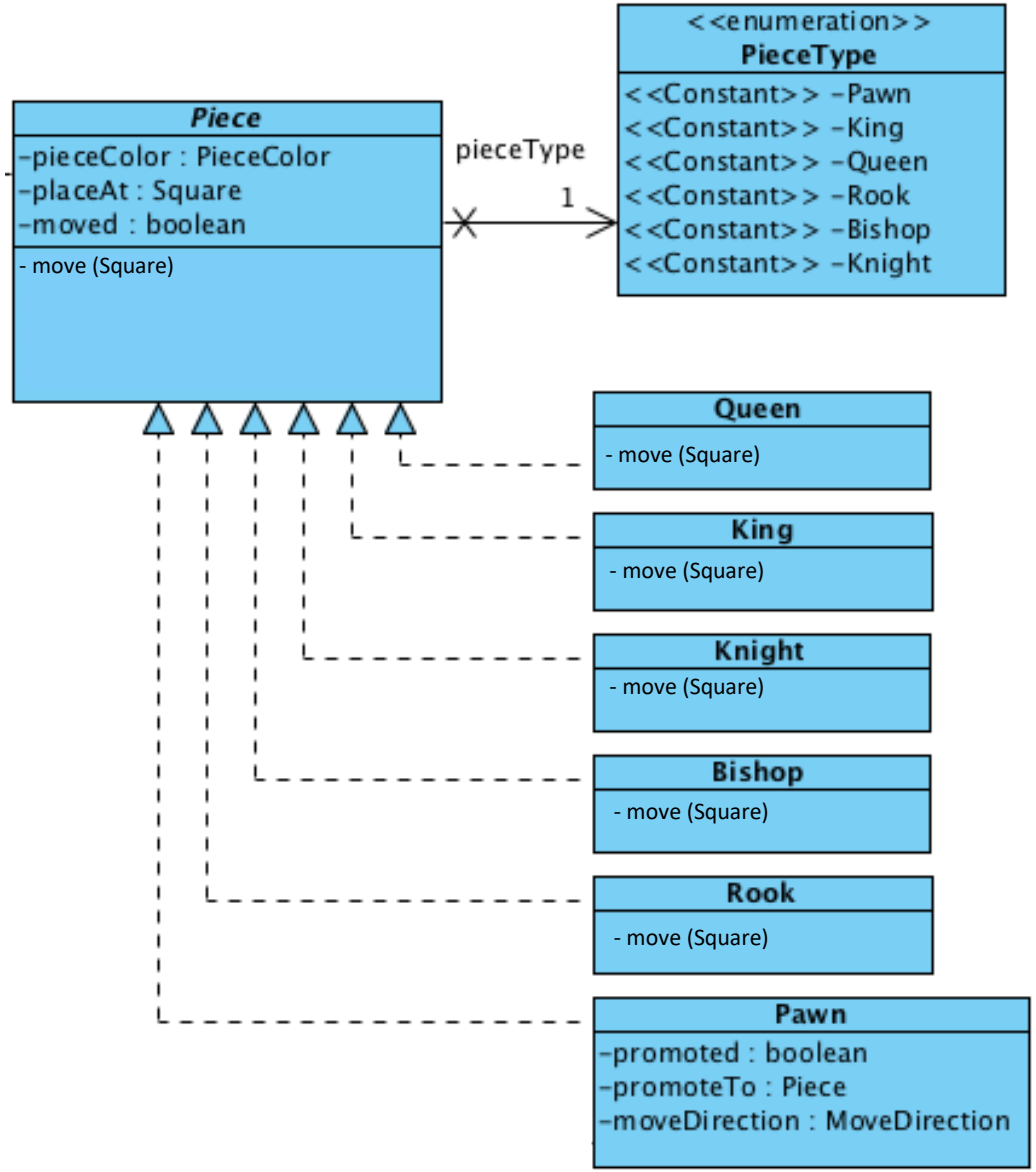
Cuidado!




```
public class Card { … }

public class MainClass {

public static void main(String[] args) {

Card Spades1 = new Card(“spades, 1”);
…
Card Diamond3 = new Card(“Diamond, 3”);
}
```

*Herencia vs Clase(diferentes parámetros)

Cuidado!



- Si es herencia si:
  - Se implementan funciones de diferente forma (polimorfismo)

Bibliografía

- Deitel, H. M. & Deitel, P. J.(2008). *Java: como programar*. Pearson education. Séptima edición.
- Sumérgete en los patrones de diseño. V2021-1.7. Alexander Shvets. https://refactoring.guru/es/design-patterns/book

  Versión online: https://refactoring.guru/es/design-patterns/catalog
- The Java tutorials. https://docs.oracle.com/javase/tutorial/
- Páginas de apoyo para la sintaxis, bibliotecas, etc.:
  - https://www.sololearn.com
  - https://www.w3schools.com/java/default.asp
  - https://docstore.mik.ua/orelly/java-ent/jnut/index.htm

Técnicas de Programación Avanzada (Java)

Curso 2021-2021

```
exit(); //Gracias!
```

Dra. Nieves Cubo Mateo