Técnicas de programación avanzada

Curso 2023-2024

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

Dra. Nieves Cubo Mateo (@Nicuma3)
Alejandro Alonso Puig (@mundobot)

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. Relaciones.

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. Relaciones.

Herencia ("is a")

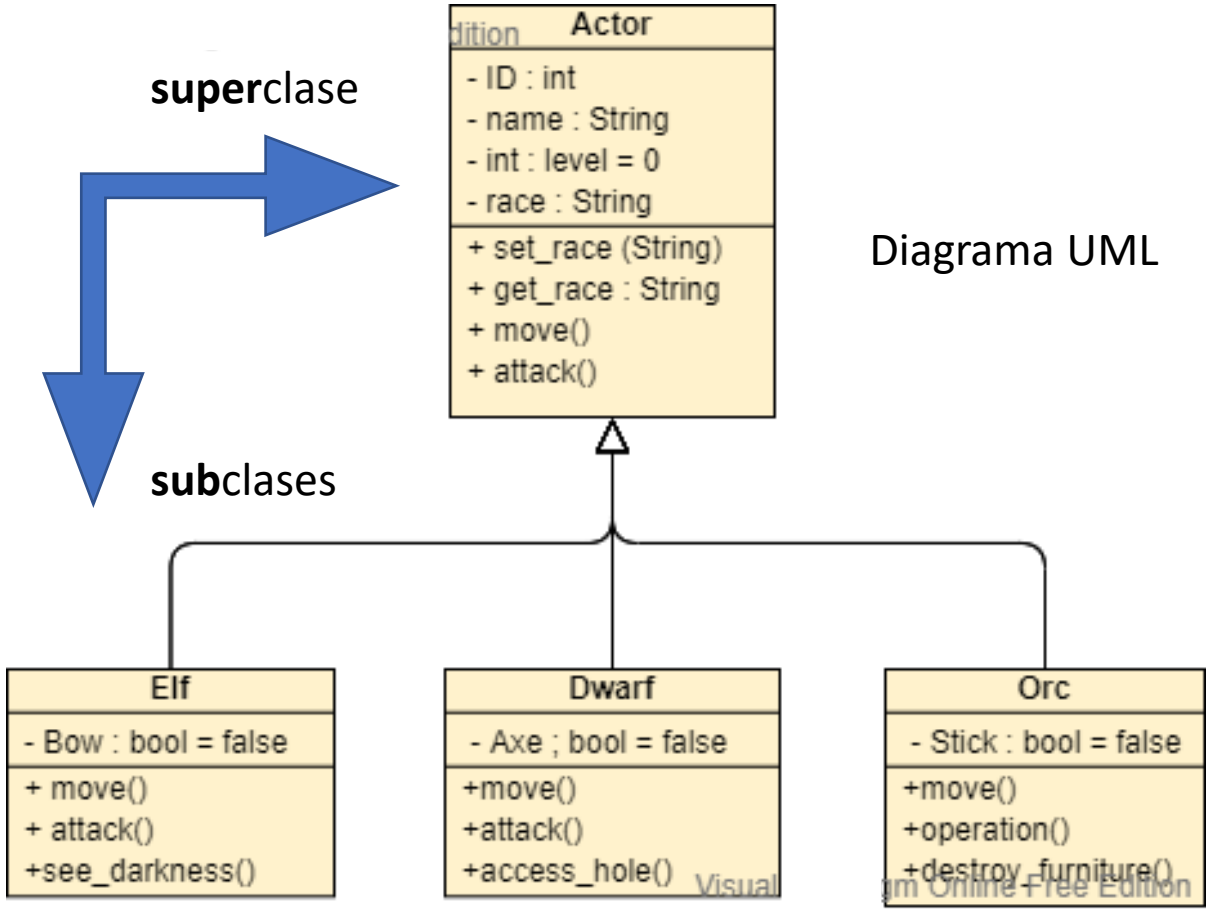


Diagrama UML

- Relación de extension. "Es un.."
- Se heredan los métodos y atributos
- Los miembros privados no son accesibles por las clases hijas
- Las clases hijas pueden añadir sus propios atributos y métodos
- Sólo se puede heredar de 1 clase
- Permite reutilizar código y agrupar objetos bajo una superclase

- **Público/Public(+):** cualquiera tiene acceso
- **Privado/Private(-):** únicamente la clase puede acceder a la propiedad o método.
- **Protegido/Protected(#):** las clases del mismo paquete y que heredan de la clase pueden acceder a la propiedad o método.
- **Paquete / Package private (~)** (valor por defecto si no se indica ninguno): solo las clases en el mismo paquete pueden acceder a la propiedad o método.
- **Derivado/Derived property (/):** producido o calculado a partir del valor de otro atributo o método
- **Estático/ static (subrayado):** Se puede acceder directamente a una variable estática por el nombre de clase y no necesita ningún objeto

class nombreSubclase **extends** nombreSuperclase{ }

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Herencia (“is a”)

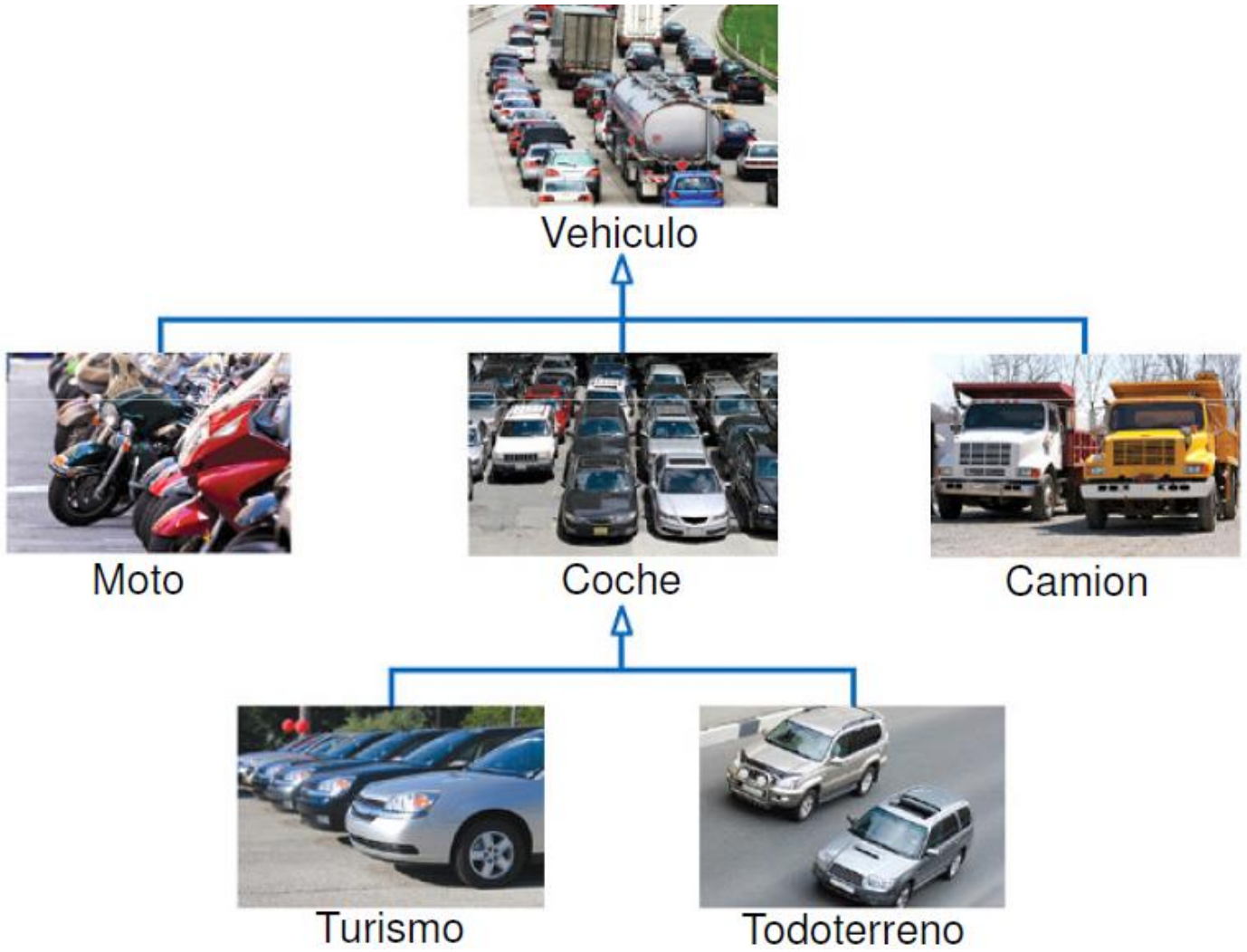
```
public class Actor {…}

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

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

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

Herencia (“is a”)



Otro ejemplo de herencia

Fuente: https://personales.unican.es/corcuerp/java/

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

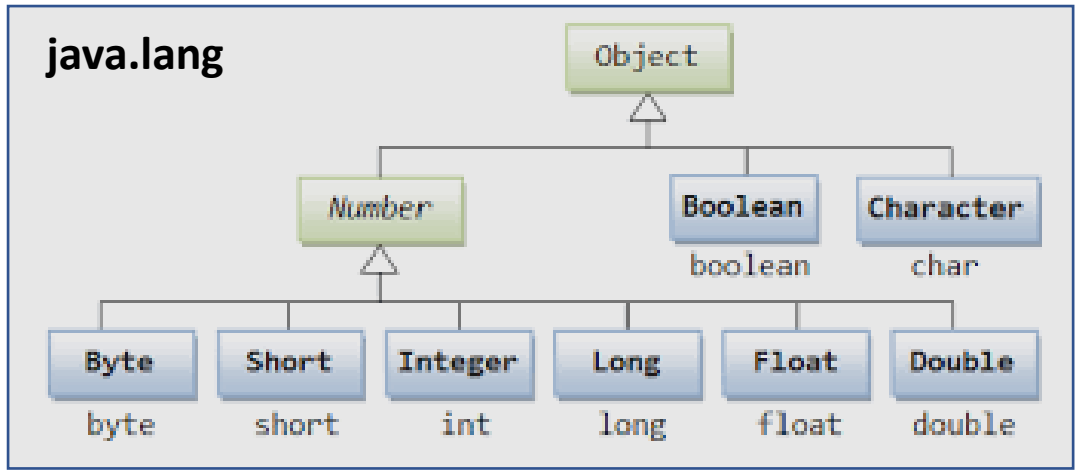
Herencia (“is a”)

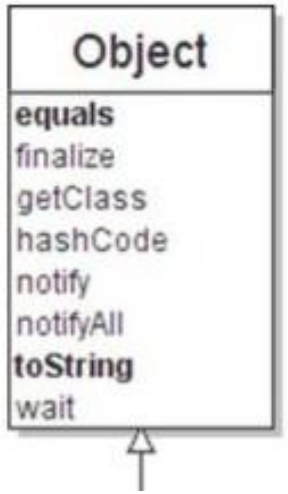
Visibilidad	Public	Protected	Default	Private
Desde la misma clase	✓	✓	✓	✓
Desde otra clase del mismo paquete	✓	✓	✓	✗
Subclase del mismo paquete	✓	✓	✓	✗
Subclase fuera del mismo paquete	✓	✓	✗	✗
Clase fuera del paquete	✓	✗	✗	✗

- Se heredan los métodos y atributos privados o protegidos
- Los miembros privados no son accesibles por las clases hijas

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Herencia ("is a")





- Paquete **java.lang** contiene las clases más importantes del lenguaje de Java.
- La clase **Object** **es la raíz de toda la jerarquía de clases en Java** (ultimate superclass). Es decir, cualquier clase es implicitamente hija de Object.
- Los datos primitivos son los únicos que no son objetos.
- Cada clase Java deriva, directa o indirectamente, de la clase Object. La clase padre inmediatamente superior a la clase que se está declarando se conoce como superclass. Si no se especifica la superclase de la que deriva una clase, se entiende que deriva directamente de la clase Object (definida en el paquete java.lang).
- En la declaración de una clase se utiliza la palabra clave *extends* para especificar la superclase, de la forma:

```
class MiClase extends SuperClase {
   // cuerpo de la clase
   }
```

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Redefinición de miembros heredados

```
public class Person {
  :
  public String getName() {
    System.out.println("Parent: getName");
    return name;
  }
}
```

```
public class Student extends Person {
  :
  public String getName() {
    System.out.println("Student:getName");
    return name;
  }
  :
}
```

Cuando se invoca el método getName de un objeto Student, el resultado es: `Student:getName`

Cuando una clase hereda una serie de miembros, en ocasiones puede interesar modificar el tipo de algún atributo o redefinir un método. Este mecanismo se conoce como:

- *Ocultación* cuando se trata de un atributo y
- *Sustitución u overriding* cuando se trata de un método

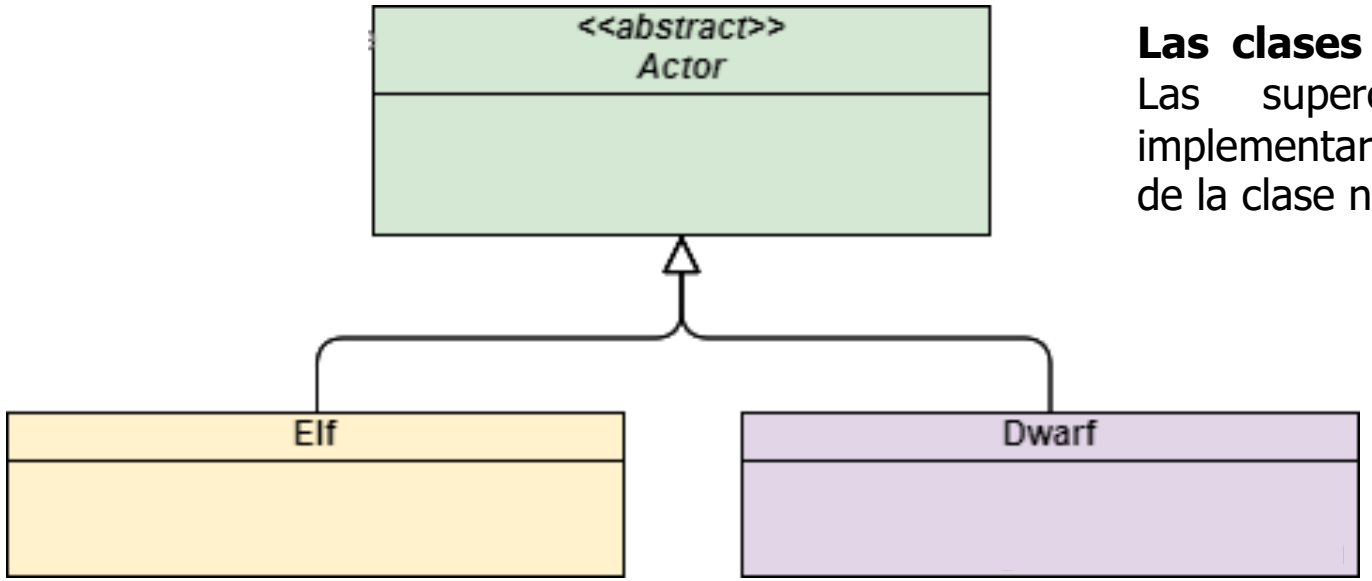
Consiste en declarar un miembro con igual nombre que uno heredado, lo que hace que este sea ocultado o sustituido por el nuevo.

Es buena costumbre, pero no obligatorio, poner "@Override" en la línea previa.

Fuente: https://personales.unican.es/corcuerp/java/

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

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. Relaciones.

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

- 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");
}
```

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Interfaces – 'Qué', pero no 'cómo'

- Las interfaces son clases puramente abstractas
- No tienen atributos (puede tener constantes: static final)
- Poseen métodos sin implementar (puede haber static final implementados)
- 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

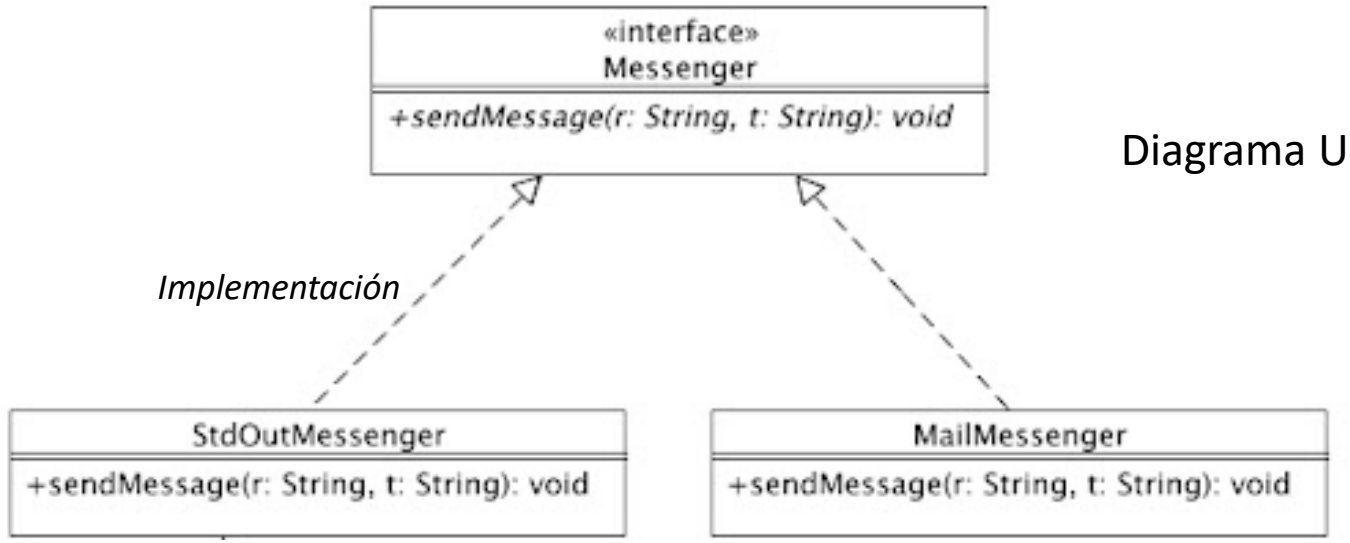


Diagrama UML

class nombreSubclase **implements** nombreInterfaz{ }

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Interfaces vs clases abstractas

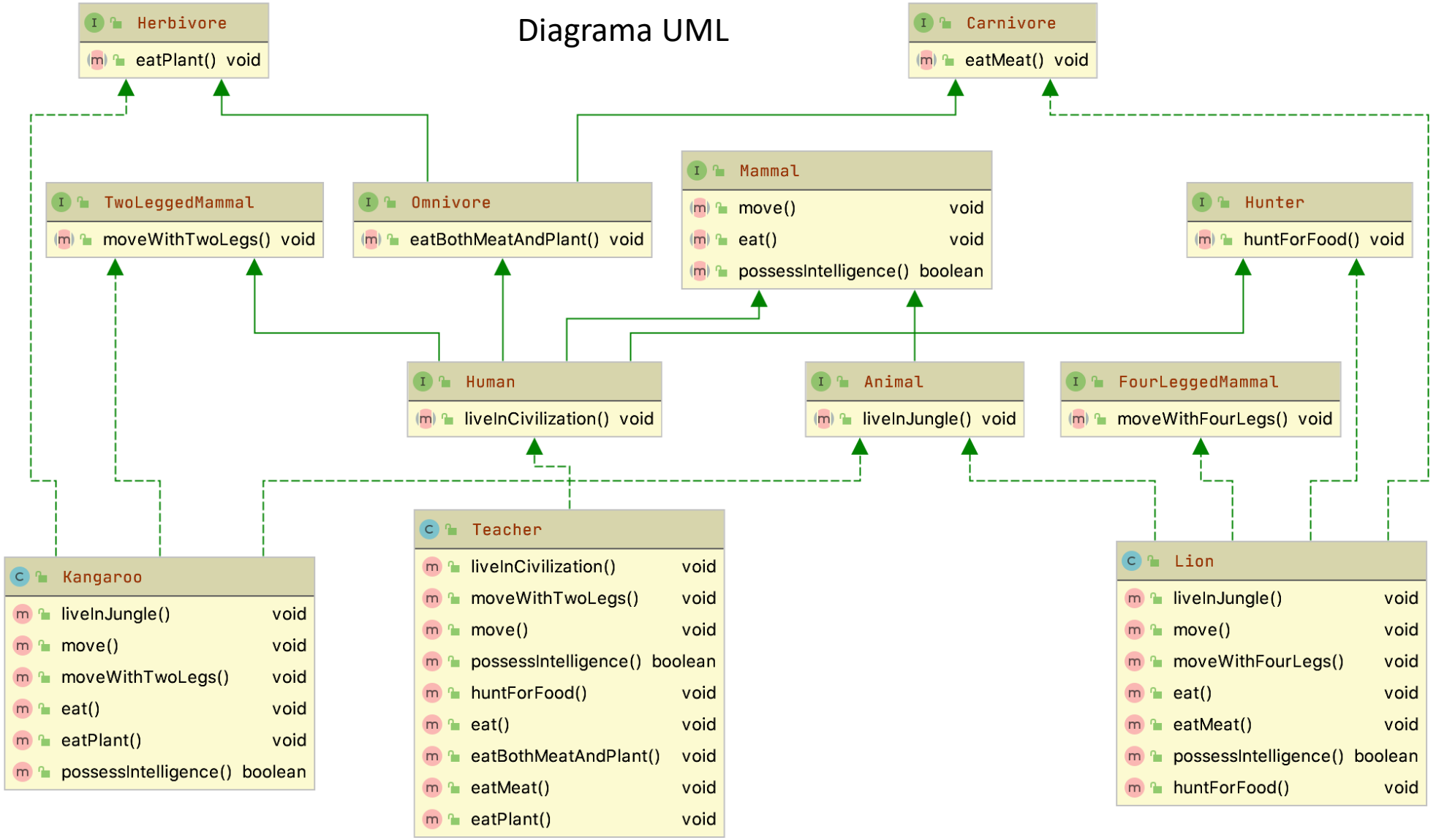
Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete 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. Relaciones.

Ejemplo de uso



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

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Ejemplo de uso

```java
public interface Mammal {
    void move();
    boolean possessIntelligence();
}
```

```java
public interface Animal extends Mammal {
    void liveInJungle();
}
```

```java
public interface Human extends Mammal, TwoLeggedMammal, Omnivore, Hunter {
    void liveInCivilization();
}
```

```java
public interface Carnivore {
    void eatMeat();
}
```

```java
public interface Herbivore {
    void eatPlant();
}
```

```java
public interface Omnivore extends Carnivore, Herbivore {
    void eatBothMeatAndPlant();
}
```

```java
public interface FourLeggedMammal {
    void moveWithFourLegs();
}
```

```java
public interface TwoLeggedMammal {
    void moveWithTwoLegs();
}
```

```java
public interface Hunter {
    void huntForFood();
}
```

```java
public class Kangaroo implements Animal, Herbivore, TwoLeggedMammal {
    @Override
    public void liveInJungle() {
        System.out.println("I live in Outback country");
    }

    @Override
    public void move() {
        moveWithTwoLegs();
    }

    @Override
    public void moveWithTwoLegs() {
        System.out.println("I like to jump");
    }

    @Override
    public void eat() {
        eatPlant();
    }

    @Override
    public void eatPlant() {
        System.out.println("I like this grass");
    }

    @Override
    public boolean possessIntelligence() {
        return false;
    }
}
```

```java
public class Lion implements Animal, FourLeggedMammal,
    @Override
    public void liveInJungle() {
        System.out.println("I am king of the jungle!")

    }

    @Override
    public void move() {
        moveWithFourLegs();
    }

    @Override
    public void moveWith
        System.out.print
    }

    @Override
    public void eat() {
        eatMeat();
    }

    @Override
    public void eatMeat(
        System.out.print
    }

    @Override
    public boolean posse
        return false;
    }

    @Override
    public void huntForF
        System.out.prin
    }
}
```

```java
public class Teacher implements Human {
    @Override
    public void liveInCivilization() {
        System.out.println("I live in an apartment");
    }

    @Override
    public void moveWithTwoLegs() {
        System.out.println("I wear shoes and walk with two legs one in front of the
    }

    @Override
    public void move() {
        moveWithTwoLegs();
    }

    @Override
    public boolean possessIntelligence() {
        return true;
    }

    @Override
    public void huntForFood() {
        System.out.println("My ancestors used to but now I mostly rely on cattle");
    }

    @Override
    public void eat() {
        eatBothMeatAndPlant();
    }

    @Override
    public void eatBothMeatAndPlant() {
        eatPlant();
```

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Relaciones

Relaciones

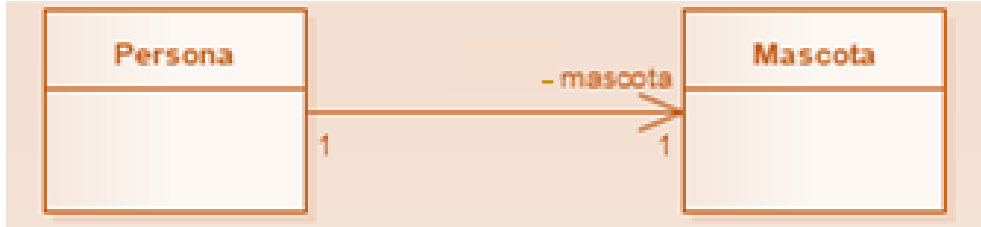
Una relación es un término general que abarca los tipos específicos de conexiones lógicas que se pueden encontrar en los diagramas de clases y objetos. UML presenta las siguientes relaciones:

Class Diagram Relationship Type	Notation
Association	
Inheritance	
Realization/ Implementation	
Dependency	
Aggregation	
Composition	

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Relaciones: Asociación

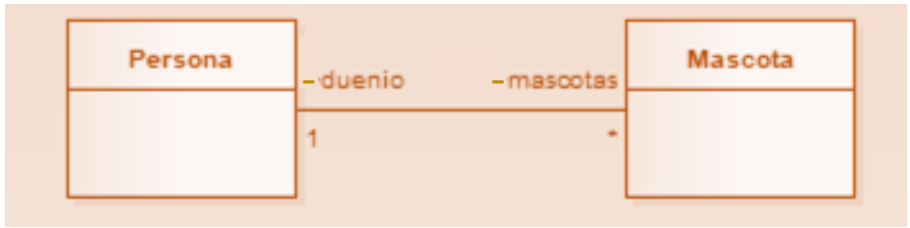
En este ejemplo tenemos a la clase Persona relacionada con la clase Mascota por medio de una relación de **asociación** con una **multiplicidad de 1 a 1**, es decir, una Persona tiene una Mascota. Observa la dirección en la que fluye la relación, va desde Persona hacia Mascota, y esto lo sabemos por la flecha que tiene del lado derecho, lo que nos indica que va en el sentido de izquierda a derecha y se lee de esa manera, pero en codificación nos hace saber que la clase Persona posee un atributo de tipo Mascota, pero SÓLO 1, ya que eso es lo que nos indica la multiplicidad.




```java
public class Persona {
    private Mascota mascota;

    public Persona(){
        //...
    }
}
```

```java
public class Mascota {
    // atributos de Mascota

    public Mascota(){
        // ...
    }
}
```

La relación también puede ser **bidireccional** (sin flecha) si se implementan atributos en ambas clases que relacionan una con la otra, e incluso con una relación de multiplicidad de 1 a muchos (*) si se utilizan arrays, listas,…



Fuente: https://www.arkanapp.com

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Relaciones: Agregación

La **agregación** es un tipo de relación que indica que un objeto forma parte o le pertenece a otro objeto, es prácticamente una asociación, pero se diferencian por la notación que se utiliza en UML y su funcionalidad dentro del código. El diamante o rombo blanco queda del lado de la clase TODO, que almacena objetos de la clase PARTE

MAIN

```java
public class ProgramaMain {
    public static void main(String[] args){
        Tienda tienda = new Tienda();
        tienda.addCliente(new Cliente("Mauricio"));
        tienda.addCliente(new Cliente("Frey"));
    }
}
```

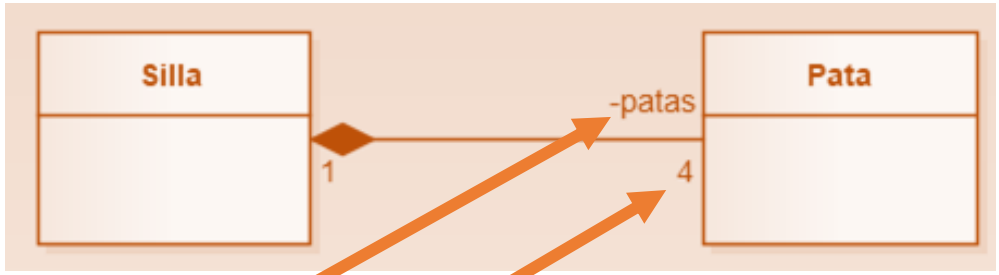
```java
public class Tienda {
    private Cliente[] clientes = new Cliente [3];
    private int numClientes = 0;

    public Tienda(){}

    public void addCliente(Cliente cliente){
        clientes[numClientes] = cliente;
        numClientes++;
    }
}
```

Fuente: https://www.arkanapp.com

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Relaciones: Composición

La **composición** es una relación como la agregación, pero más fuerte, es decir, un objeto no puede ser ese objeto sin otros objetos, por ejemplo: una silla no puede ser silla sin sus patas, un automóvil no puede ser automóvil sin sus ruedas o su motor, básicamente todos dependen de entre sí.



El constructor de Silla crea la instancia de Pata

MAIN

```java
public class ProgramaMain {
    public static void main(String[] args){
        Silla silla = new Silla();
        silla.agregarPata("Negro", 10f);
        silla.agregarPata("Negro", 10f);
        silla.agregarPata("Negro", 10f);
        silla.agregarPata("Negro", 10f);
    }
}
```

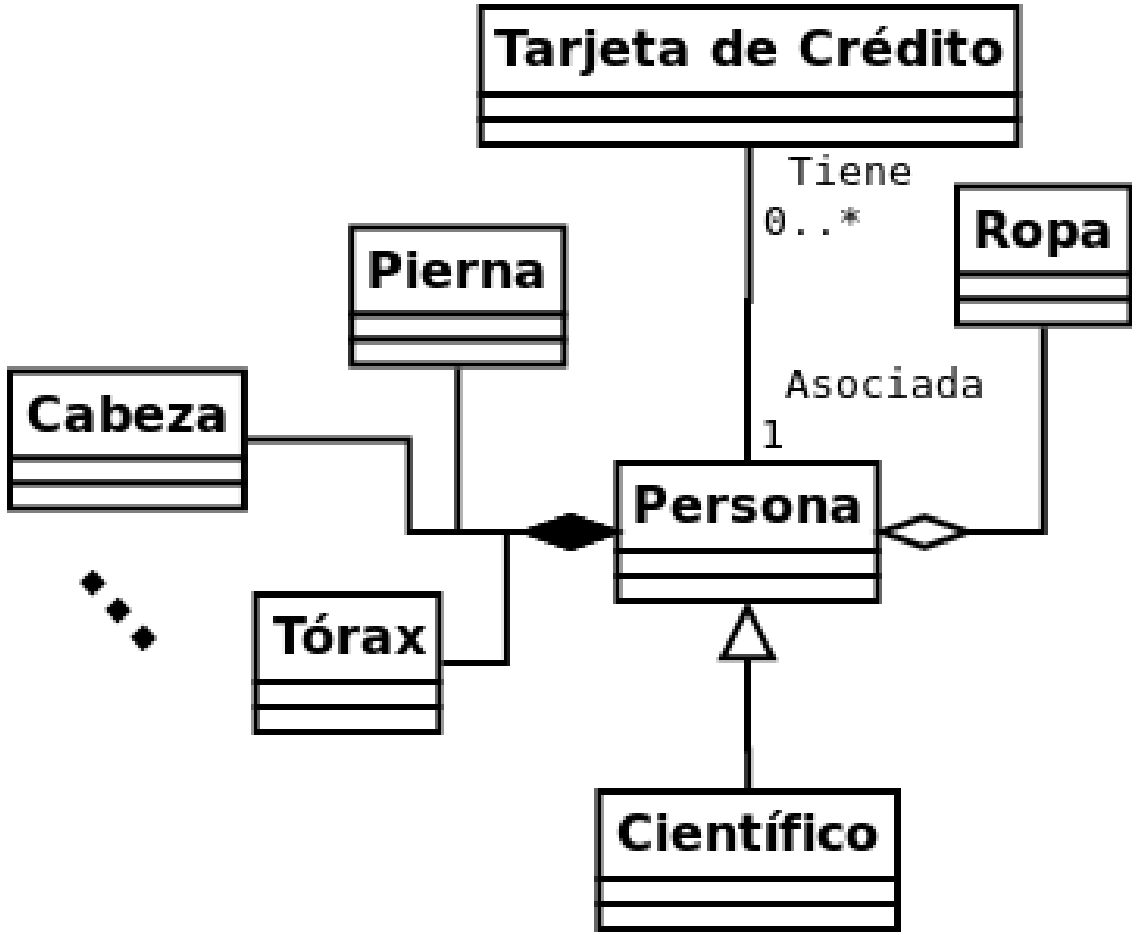
```java
public class Silla {
    private Pata[] patas;
    private int numPatas = 0;

    public Silla(){
        patas = new Pata[4];
    }

    public void agregarPata(String color, float peso){
        patas[numPatas] = new Pata(color, peso);
        numPatas++;
    }
}
```

```java
public class Pata {
    private String color;
    private float peso;

    public Pata(String color, float peso){
        this.color = color;
        this.peso = peso;
    }
}
```

Fuente: https://www.arkanapp.com

2.1 Herencia. Interfaces y clases abstractas. Relaciones.

Relaciones: Resumen



El siguiente diagrama de clases UML presenta los cuatro tipos de relaciones más comunes que se dan entre clases:

- **Herencia** (entre las clases Persona y Científico).
- **Composición** (entre la clase Persona y las clases Pierna, Cabeza y Tórax entre otras tantas posibles).
- **Agregación** (entre las clases Persona y Ropa).
- **Asociación** (entre las clases Persona y Tarjeta de Crédito).

Ejercicios

2.1 Herencia. Interfaces y clases abstractas. Relaciones

Ejercicios

1. [**Herencia**] Escriba un programa Java para crear una clase llamada Vehicle con un método llamado drive(). Cree una subclase llamada Coche que anule el método drive() para imprimir "Reparando un coche".
2. [**Clases abstractas**] Escriba un programa Java para crear una clase abstracta Animal con métodos abstractos eat() y sleep(). Cree subclases Lion, Tiger y Deer que amplíen la clase Animal e implementen los métodos eat() y sleep() de manera diferente en función de su comportamiento específico.
3. [**Interfaces**] Escriba un programa Java para crear una interfaz jugable con un método play() que no acepte argumentos y devuelva void. Cree tres clases Fútbol, Voleibol y Baloncesto que implementen la interfaz jugable y anulen el método play() para jugar los deportes respectivos.

2.2 Polimorfismo

Uso

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

En programación orientada a objetos, polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación. Dicho de otro modo, el objeto como entidad puede contener valores de diferentes tipos durante la ejecución del programa.

Ejemplo clásico de poliformismo. Podemos crear dos clases distintas: Gato y Perro, que heredan de la superclase Animal. La clase Animal tiene el método abstracto makesound() que se implementa de forma distinta en cada una de las subclases (gatos y perros suenan de forma distinta). Entonces, un tercer objeto puede enviar el mensaje de hacer sonido a un grupo de objetos Gato y Perro por medio de una variable de referencia de clase Animal, haciendo así un uso polimórfico de dichos objetos

```
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();
      }
   }
}
```

Ejercicios

2.2 Polimorfismo

Ejercicios

1. [**Polimorfismo**] Escriba un programa Java para crear una clase Vehículo con un método llamado speedUp(). Crea dos subclases Coche y Bicicleta. Anule el método speedUp() en cada subclase para aumentar la velocidad del vehículo de manera diferente.

2.3 Gestión de Excepciones.

Concepto

Los fallos en los programas son prácticamente inevitables. Por ejemplo un fichero al que pretendemos acceder pero el fichero ya no existe puede generar una parada del programa.

```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)
```

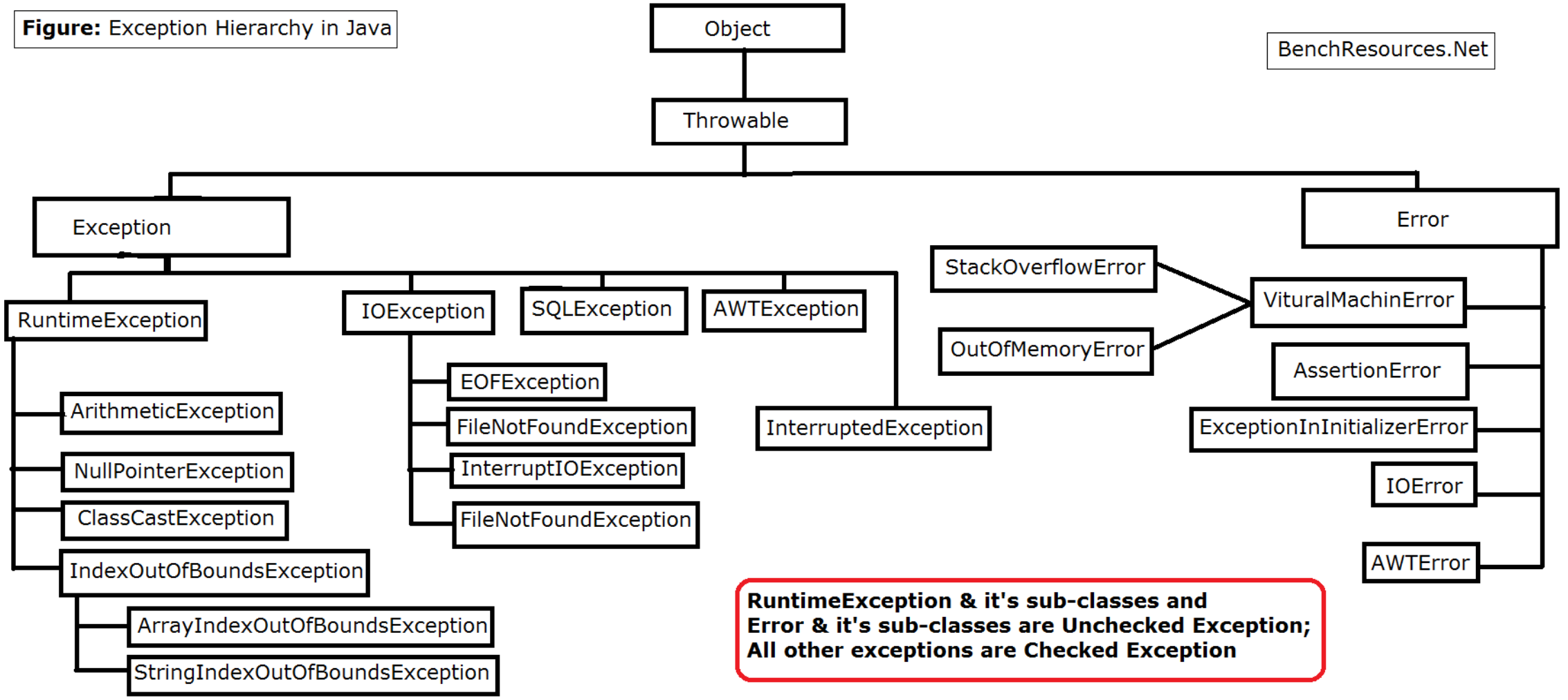
2.3 Gestión de Excepciones

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, ArithmeticException</td><td>FileNotFoundException, …</td><td>OutOfMemoryError ,IOError</td></tr>
</table>

2.3 Gestión de Excepciones

Concepto



2.3 Gestión de Excepciones.

Concepto

```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 aritmético");
     }
     finally{
        System.out.println("FIN");
     }
   }
}
```

```
Error aritmético
FIN
```

Java permite tener cierto control sobre este tipo de situaciones mediante el uso de **Excepciones**, que son herramientas para capturar de forma controlada el flujo del programa y así poder tomar acciones en consecuencia. Aquí entran en juego términos como:

- **try**: Para controlar la ejecución de una parte del código susceptible de generar una excepción
- **catch**: Para capturar una excepción de un tipo concreto y tomar acciones en consecuencia
- **finally**:

  El código del bloque `finally` se ejecuta siempre una vez que se entra en un bloque try, incluso en:
  - Salidas forzadas que usan return, continue o break
  - Terminación normal
  - Captura de excepción lanzada
  - Excepción no capturada

- **throws**: Para controlar una excepción en un método (*ver siguiente slide*)

2.3 Gestión de Excepciones.

Concepto

- Excepciones permiten escribir el flujo principal del código y tratar los casos excepcionales aparte

```
readFile {
  try {
    abrir el fichero;
    determinar el tamaño;
    reservar memoria;
    leer el fichero en memoria;
    cerrar el fichero; }
  catch (fileOpenFailed) { doSomething; }
  catch (sizeDeterminationFailed) { doSomething;}
  catch (memoryAllocationFailed) { doSomething; }
  catch (readFailed) { doSomething; }
  catch (fileCloseFailed) { doSomething; } }
```

Las excepciones no evitan el esfuerzo de detectar, reportar y gestionar los errores, pero ayuda a organizar el código de forma más efectiva

Ejemplo

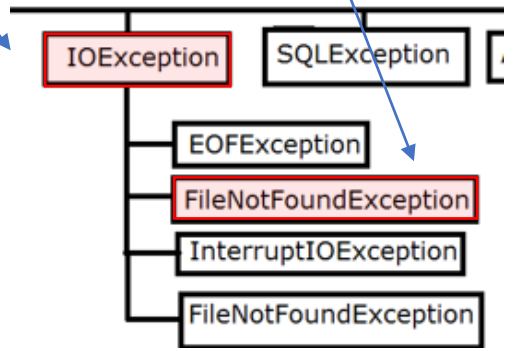
```
class MultipleCatch {
  public static void main(String args[]) {
    try {
      int den = Integer.parseInt(args[0]);
      System.out.println(3/den);
    } catch (ArithmeticException exc) {
      System.out.println("Divisor 0");
    } catch (ArrayIndexOutOfBoundsException exc2) {
      System.out.println("Fuera de rango");
    }
    System.out.println("Despues excepcion");
  }
}
```

- Un método puede tener manejadores específicos que pueden gestionar una excepción específica

```
catch (FileNotFoundException e) {
...
}
```

- Otra alternativa es capturar una excepción general

```
// Captura todas las excepciones I/O, incluyendo
// FileNotFoundException, EOFException, etc.
catch (IOException e) {
...
}
```



Fuente: https://personales.unican.es/corcuerp/java/

2.3 Gestión de Excepciones.

Lanzamiento de excepciones

Lanzar excepciones

- Java permite el lanzamiento o generación de excepciones

  `throw <objeto excepcion>;`

- La excepción que se genera es un objeto, por lo que hay que crearlo como cualquier otro objeto
- Ejemplo:

  `throw new ArithmeticException("probando...");`

```
class ThrowDemo {
  public static void main(String args[]){
    String input = "input invalido";
      try {
        if (input.equals("input invalido")) {
          throw new RuntimeException("throw demo");
        } else {
          System.out.println(input);
        }
        System.out.println("Despues throwing");
      } catch (RuntimeException e) {
        System.out.println("Excepcion capturada:" + e);
      }
  }
}
```

Fuente: https://personales.unican.es/corcuerp/java/

2.3 Gestión de Excepciones.

Lanzamiento de excepciones

```java
package pruebas;
public class PersonaAdulta {
    private String nombre;
    private int edad;

    public PersonaAdulta(String nombre, int edad) throws Exception {
        this.nombre = nombre;
        if (edad < 18)
            throw new Exception("No es adulta la persona " + nombre + " porque tiene " + edad + " años.");
        this.edad = edad;
    }

    public void fijarEdad(int edad) throws Exception {
        if (edad < 18)
            throw new Exception("No es adulta la persona " + nombre + " porque tiene " + edad + " años.");
        this.edad = edad;
    }

    public void imprimir() {
        System.out.println(nombre + " - " + edad);
    }

    public static void main(String[] ar) {
        try {
            PersonaAdulta persona1 = new PersonaAdulta("Ana", 50);
            persona1.imprimir();
            PersonaAdulta persona2 = new PersonaAdulta("Juan", 13);
            persona2.imprimir();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

- Si un método puede causar una exepción (checked) pero que no lo captura, debe indicarlo mediante `throws`
- Sintaxis:

```
<tipo> <nombreMetodo> (<listaParametros>)
  throws <listaExcepciones> {
    <cuerpoMetodo>
}
```

Fuente: https://personales.unican.es/corcuerp/java/

2.3 Gestión de Excepciones

Lanzamiento de excepciones

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

Ejemplo:

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

```
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");
}
```

2.3 Gestión de Excepciones

Ejemplo:

```
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);}
```

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

2.3 Gestión de Excepciones

Creación de excepciones propias

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

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

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

Ejercicios

2.3 Gestión de Excepciones

Ejercicios

1. [Excepciones comunes] Escriba un programa Java para crear un método que tome un número entero como parámetro y genere una excepción si el número es impar.
2. [Excepciones propias] Escriba un programa Java para crear un método que tome una cadena como entrada y genere una excepción si la cadena no contiene vocales.

2.4 Genericidad y Plantillas

Concepto

En este tema se trata la genericidad, que consiste en poder utilizar el mismo código sobre distintos tipos. Es una clase de polimorfismo, pero que se resuelve en tiempo de compilación, no en tiempo de ejecución.

Ejemplo:

```
ArrayList<String> v = new ArrayList<String>();  // Creación "clásica"
ArrayList<T>      v = new ArrayList<T>();       // Creación "generica"
ArrayList<T>      v = new ArrayList<> ();       // Operador diamante
```

A partir de Java 7 está disponible el llamado operador diamante, que hace que no se exija el repetir a la derecha de la asignación, tras new, lo mismo que se ha escrito a la izquierda: en su lugar se deja la especificación del tipo vacía: "<>"

2.4 Genericidad y Plantillas

Genericidad - 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);
    }
}
```

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

Sobrecarga de métodos para aceptar diferentes tipos de datos.

Repetición de Código…
Poco eficiente….

```java
public class prueba {
    static <T> void printThisArray(T[] data) {
        for (T item:data) {
            System.out.println(item);
        }
    }
    public static void main(String[] args) {
        Integer[] arr = {3,4,5};
        printThisArray (arr);
    }
}
```

2.4 Genericidad y Plantillas

Genericidad - Clases

```
public class Pair<T> {                  //Clase genérica
    private T left, right;              //Variables genericas
    public Pair(T left, T right) { //Constructor
        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());
}
```

Se obtiene una clase con tipos genéricos.

Es importante tener en cuenta que, en el caso de utilizar tipos primitivos, como int, double, float o boolean, **es necesario utilizar sus clases asociadas** correspondientes, Integer, Double, Float y Boolean respectivamente

Ejercicios

2.4 Genericidad y Plantillas

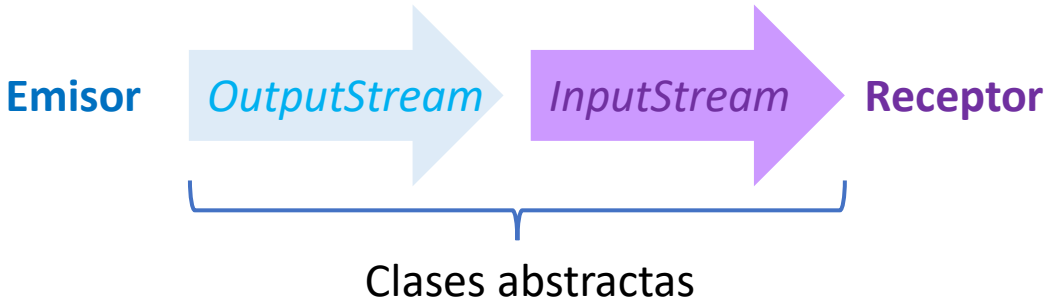
Ejercicios

1. Escriba un programa Java para crear un método genérico que tome dos matrices del mismo tipo y verifique si tienen los mismos elementos en el mismo orden.
2. Escriba un programa Java para crear un método genérico que tome una lista de números y devuelva la suma de todos los números pares e impares.

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

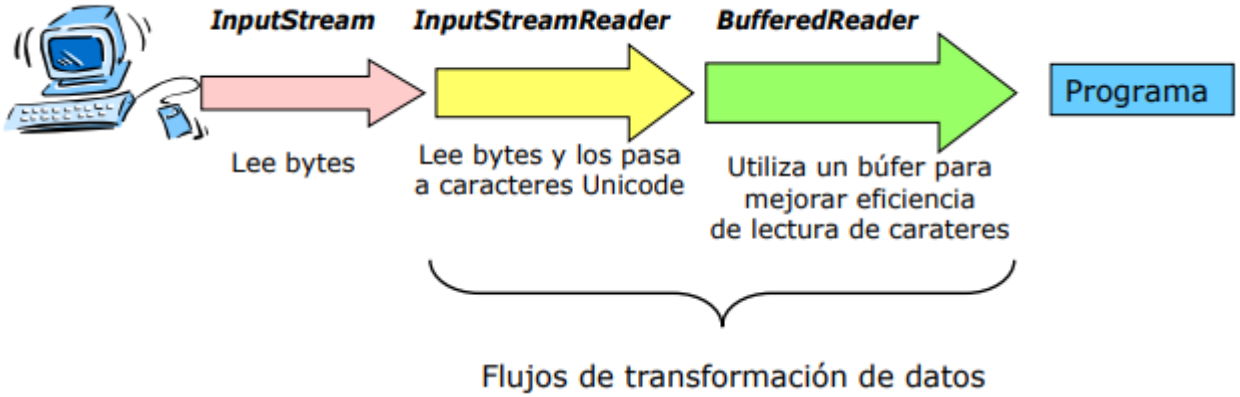
Cambio de salida standard: *System.setOut*

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.

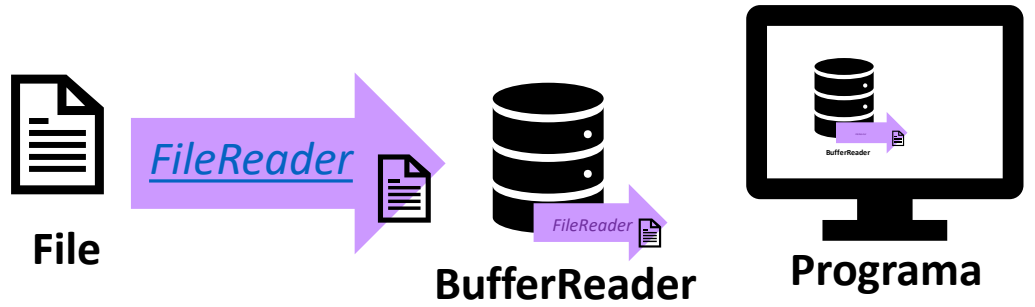
Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.
• Si vamos escribiendo, se guardará los datos hasta que tenga bastantes datos como para hacer la escritura 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

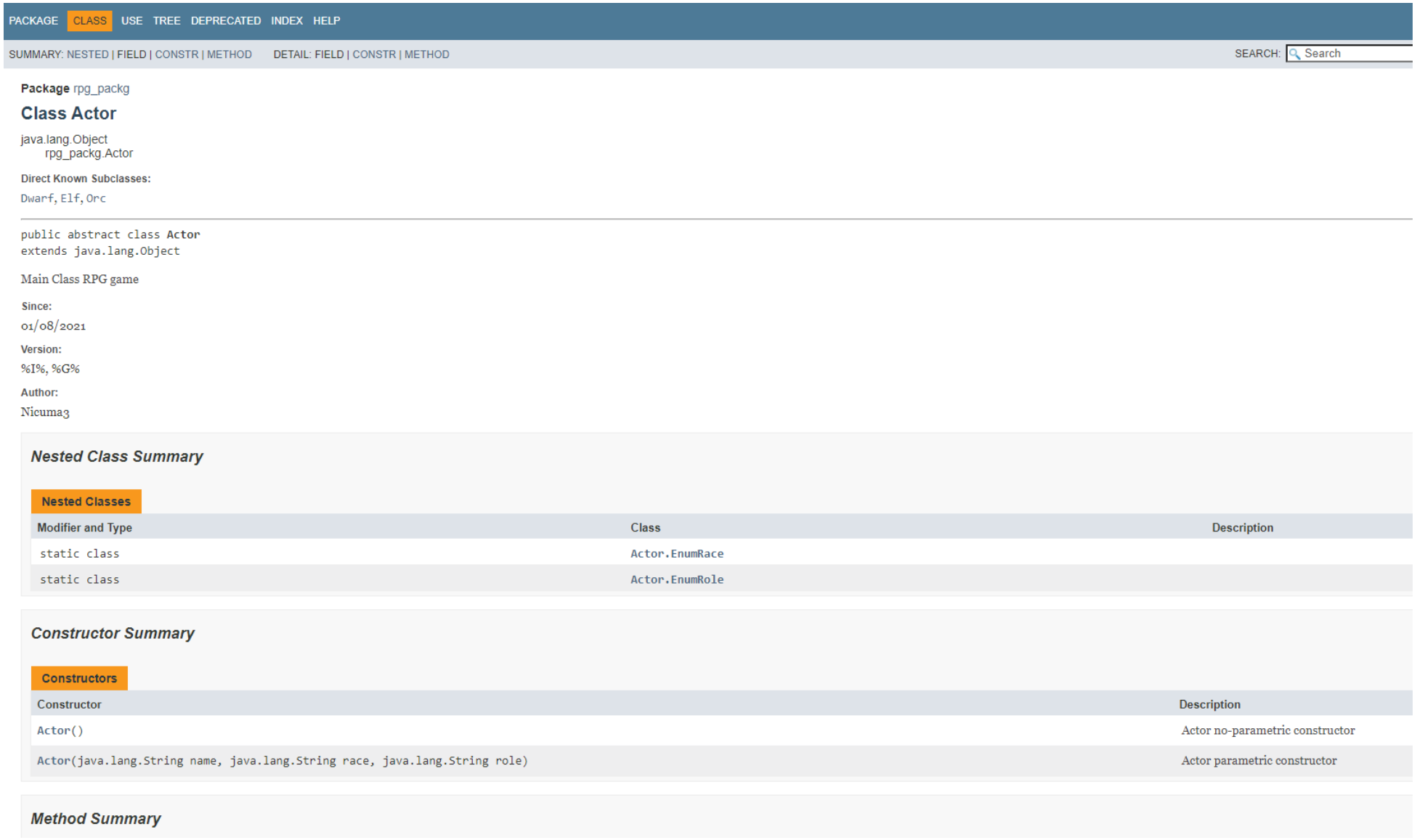
Ejemplo

```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

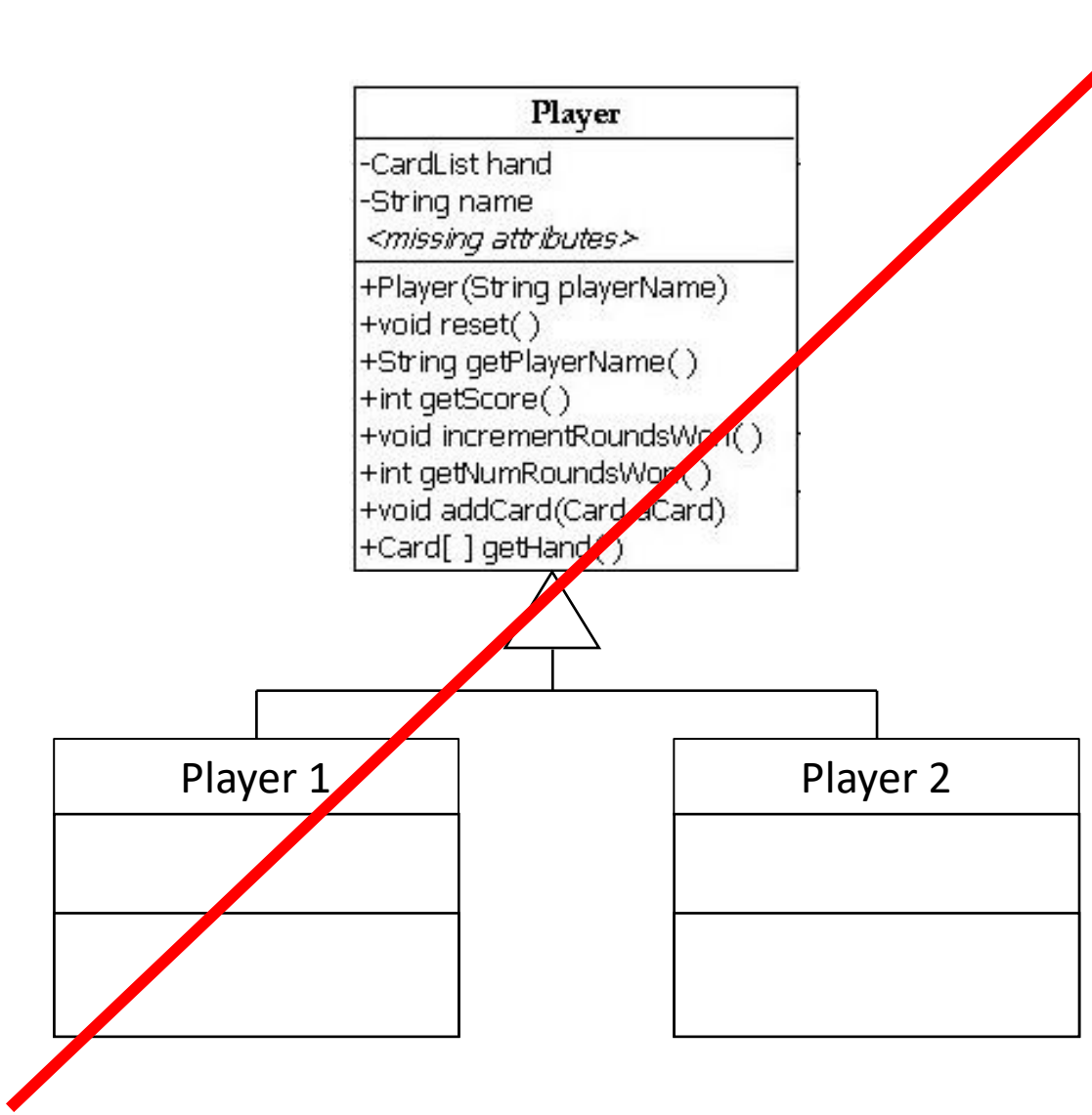
Modificadores

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

*Clase vs Objeto.

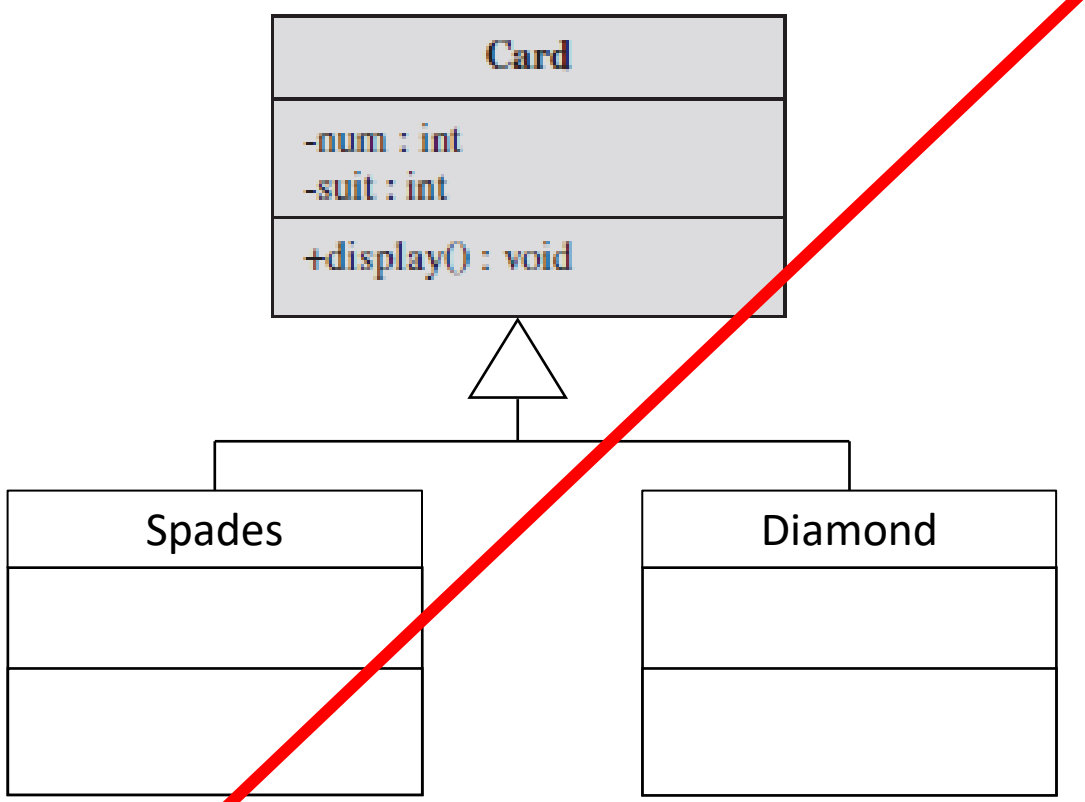
Cuidado!




```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)

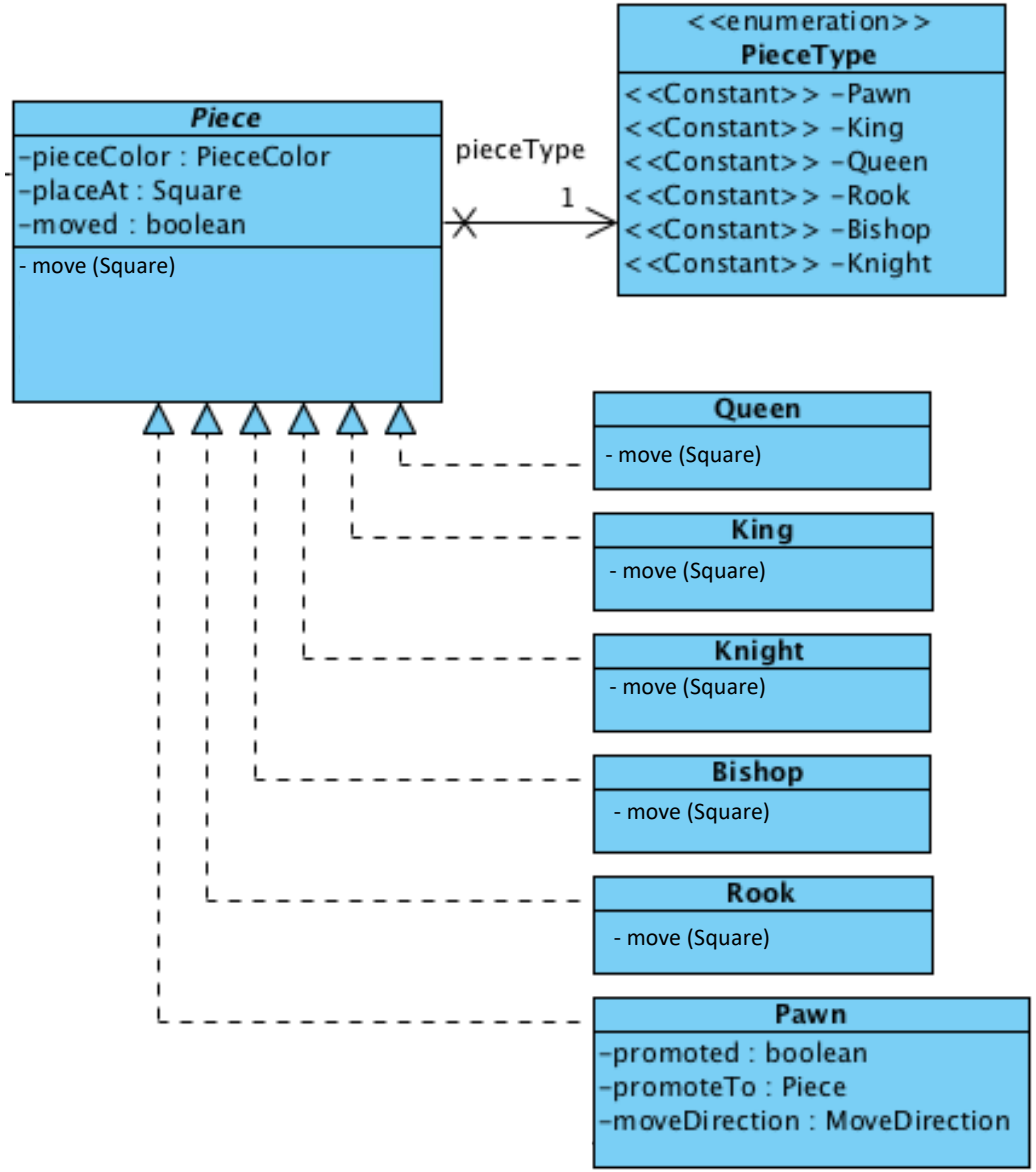
Cuidado!




```
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

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