FUNDAMENTOS DE PROGRAMACIÓN

2 – Diseño de tipos

UNIVERSIDAD DE SEVILLA

¿Qué se ve en este tema?

NamedTuple
parser_aux
~~función_1~~
~~función_2~~
~~…~~
~~función_n~~
~~función_aux1~~
…

tipo.py

TipoAux1.java

…

TipoAuxN.java

Tipo.java

¿Qué se ve en este tema?

```python
Estudiante = NamedTuple('Estudiante', (('nombre', str), ('edad', int),
...))
```

```java
public class Estudiante {
        private String nombre;
        private Integer edad;
        ...
        public Estudiante(String nombre, Integer edad, ) {
                this.nombre = nombre;
                this.edad = edad;
                ...
        }
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
}
```

ÍNDICE

1. Esquema de diseño de tipos

- Un buen diseño de tipos es básico para que los programas sean comprensibles y fáciles de mantener. Nuestra plantilla para el diseño de un tipo será la siguiente:
  - **Nombre del Tipo**
    - Propiedades:
      - Nombre de propiedad: Tipo, Consultable o no, …
      - …
    - Constructores:
      - Constructor 1
      - …
    - Restricciones:
      - Restricción 1
      - …
    - Operaciones:
      - …
    - Criterio de igualdad
    - Representación como cadena
    - Orden natural (si lo tiene)

```java
// Java bean for Person
public class Person {
    // Private variable
    private String fullName;
    // Constructor
    public Person(String fullName) {
        setFullName(fullName);
    }
    // Getter and setter for variable
    public String getFullName() {
        return fullName;
    }
    public void setFullName (String fullName) {
        this.fullName=fullName;
    }
}
```

- En este esquema, el criterio de igualdad y la representación como cadena están relacionados con el tipo *Object*, mientras que el orden natural está relacionado con el tipo *Comparable*. Estos tipos los veremos con más detalles en los siguientes apartados.

ÍNDICE

2. Clases - General

Properties

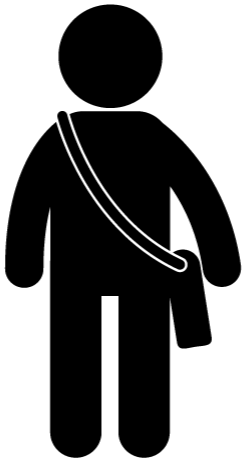
nombre
edad
dni
grupo
nota1
nota2
nota3

Methods

unirseAGrupo()
verNotas()
solicitarErasmus()
calcularNotaFinal()

Class

Estudiante



Objects

estudiante1
- "Pilar"
- 19
- 12345678A
- G1
- 4.25
- 7.5
- 9.75

estudiante2
- "Javier"
- 20
- 87654321Z
- G2
- 6.75
- 4.5
- 8.75

estudiante3
- "Alejandro"
- 18
- 18273645H
- G2
- 6.00
- 7.00
- 6.75

2. Clases

- Las **clases** son las unidades de la POO que permiten
  - Definir los detalles del **estado interno** de un objeto (mediante los **atributos**),
  - Calcular las **propiedades** de los objetos a partir de los atributos e implementar las **funcionalidades** ofrecidas por los objetos (a través de los **métodos**)
- Para implementar una clase partimos de la interfaz o interfaces que un objeto debe ofrecer y que han sido definidas previamente.
- En la clase se dan los detalles del estado y de los métodos.
- Decimos que la clase implementa (representado por la palabra **implements**) la correspondiente interfaz (o conjunto de ellas).

2. Clases - Estructura

- El patrón lo tenemos que completar con el nombre de la clase, y, de manera opcional, una serie de modificadores, una clausula ***extends***, una clausula ***implements*** y una serie de atributos y métodos.

```
[Modificadores] class NombreClase [extends ...] [implements ...] {
   [atributos]
   [métodos]
}
```

2. Clases - Atributos

- Los atributos sirven para establecer los detalles del **estado interno** de los objetos. Son un conjunto de variables, cada una de un tipo, y con determinadas restricciones sobre su visibilidad exterior.

```
[Modificadores] tipo identificador [ = valor inicial ];
```

```
private String nombre;
private Integer edad;
```

2. Clases - Métodos

- Los métodos son unidades de programa que indican la forma concreta de consultar o modificar las propiedades de un objeto determinado.

```
[Modificadores] TipoDeRetorno nombreMétodo ([parámetros formales]) {
        [Cuerpo]
}
```

```
public String getNombre(){
        return this.nombre;
}

public void setNombre(String n){
        this.nombre = n;
}

public Double calcularNotaFinal(){
        return (this.nota1 + this.nota2 + this.nota3) / 3;
}
```

2. Clases - Métodos

- De forma general, hay dos tipos de métodos: **observadores** y **modificadores**.
  - Los métodos observadores devuelven el valor del atributo al que hace referencia. Estos métodos no modifican los atributos. Normalmente serán métodos observadores aquellos cuyo nombre empiece por ***get***.
  - Los métodos modificadores modifican el estado del objeto. Normalmente todos los métodos que empiecen por ***set*** son modificadores.

```
public void setEdad(Integer edad) {      // Cabecera
       this.edad = edad;                                  // Cuerpo
}
public Double getEdad() {                 // Cabecera
       return edad;                                       // Cuerpo
}
```

2. Clases - Constructores

- Son métodos que tienen el mismo nombre que la correspondiente clase.
- Sirven para crear objetos nuevos y establecer el estado inicial de los objetos creados.

```
public Estudiante(String nombre, Integer edad, String dni, Grupo grupo, Double
nota1, Double nota2, Double nota3) {
    this.nombre = nombre;
    this.edad = edad;
    this.dni = dni;
    this.grupo = grupo;
    this.nota1 = nota1;
    this.nota2 = nota2;
    this.nota3 = nota3;
}
```

2. Clases - Constructores

- No tienen que recibir siempre todos los atributos como parámetros
- Puede haber propiedades que se inicialicen con un valor por defecto

```
public Estudiante(String nombre, Integer edad, String dni, Grupo grupo) {
   this.nombre = nombre;
   this.edad = edad;
   this.dni = dni;
   this.grupo = grupo;
   this.nota1 = 0.0;
   this.nota2 = 0.0;
   this.nota3 = 0.0;
}
```

2. Clases - Interfaces

- Una interfaz es un elemento de la POO que permite:
  - Establecer **cuáles** son las propiedades
  - ***Qué*** *se puede hacer con un objeto de un determinado tipo*, pero no se preocupa de saber **cómo** se hace.
- Formalmente una interfaz (**interface**) contiene, principalmente, las **signaturas** de los métodos que nos permiten consultar y/o cambiar las propiedades de los objetos, así como del resto de operaciones que definen la funcionalidad.

```
public interface IEstudiante {
   String getNombre();
   void setNombre(String nombre);
   Integer getEdad();
   void setEdad(Integer edad);
   String getDni();
   void setDni(String dni);
   …
}
```

2. Clases - Ejercicios

- Crea las siguientes clases, interfaces y constructores parar crear los siguientes objetos. Ten en cuenta el tipo de dato de los atributos:

1. Jugador de fútbol: nombre, fecha de nacimiento, altura, nacionalidad.
2. Videojuego: nombre, distribuidora, año de lanzamiento, ventas globales.
3. Empresa: nombre, CIF, fecha de fundación, número de empleados.
4. País: nombre, número de habitantes, extensión, capital.
5. Artículo de tienda: nombre, referencia, precio, categoría, stock.

ÍNDICE

3. Herencia

- La **herencia** es una propiedad por la que se establecen relaciones que podemos denominar **padre-hijo entre interfaces o entre clases.**
- Se representa mediante la cláusula **extends.**
- En Java la herencia entre interfaces puede ser múltiple: un hijo puede tener uno o varios padres. La interfaz hija tiene todas las signaturas (métodos) declaradas en los padres. No se puede ocultar ninguna.


```
[Modificadores] class NombreClaseHija extends ClsePadre [,...] {
}
```

```
public class Estudiante extends Persona {
      [signaturas de métodos]
}
```

ÍNDICE

4. Tipo Object

- En Java existe una clase especial llamada *Object*.
- Todas las clases heredan de *Object*, es decir, implícitamente *Object* es un tipo al que extienden todas las clases Java.
- Como cualquier tipo, *Object* proporciona una serie de métodos públicos. Aunque tiene más métodos, en este tema nos vamos a centrar solamente en:
  - El método ***equals(Object o)*** se utiliza para decidir si el objeto es igual al que se le pasa como parámetro.
  - El método ***hashCode()*** devuelve un entero, que es el código *hash* del objeto. Todo objeto tiene, por lo tanto, un código *hash* asociado.
  - El método ***toString()*** devuelve una cadena de texto que es la representación exterior del objeto. Cuando el objeto se muestre en la consola tendrá el formato indicado por su método *toString* correspondiente.

4. Tipo Object


```
@Override
public int hashCode() {
            return Objects.hash(dni, nombre);
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
                return true;
    if (obj == null)
                return false;
    if (getClass() != obj.getClass())
                return false;
    Estudiante other = (Estudiante) obj;
    return Objects.equals(dni, other.dni) && Objects.equals(nombre, other.nombre);
}

public String toString() {
            return "Estudiante [nombre=" + nombre + ", edad=" + edad + ", dni=" + dni + ", nota1=" + nota1 + ", nota2="
+ nota2 + ", nota3=" + nota3 + "]";
}
```

4. Tipo Object
Igualdad e Identidad

- El operador de **igualdad** en Java es **==**. Devuelve un tipo *boolean* y toma dos operandos del mismo tipo.
- Dos objetos son **iguales** cuando los valores de sus propiedades observables son iguales.
- Dos objetos son **idénticos** cuando al modificar una propiedad observable de uno de ellos, se produce una modificación en la del otro y viceversa.

```
int i = 7;
int j = 4;
int k = j;
boolean a = (i == j); // a es false
boolean b = (k == j); // b es true
System.out.println(a);
System.out.println(b);
```

4. Tipo Object
Igualdad e Identidad

- Para decidir si dos objetos son iguales utilizamos el método equals, que se invoca mediante la expresión o1.*equals(o2) y* que devuelve un valor tipo *boolean*:

```
Punto p1 = new PuntoImpl(1.0, 2.0);
Punto p2 = new PuntoImpl(1.0, 2.0);

boolean c = (p1 == p2); // c es false
boolean d = p1.equals(p2); // d es true
```

4. Tipo Object
Igualdad e Identidad


```
Punto p1 = new PuntoImpl(1.0, 1.0);
Punto p2 = new PuntoImpl(3.0, 1.0);
Punto p3 = p1;
p1.setX(3.0);

boolean a = (p3 == p1); // a es true
boolean b = (p3 == p2); // b es false
double x3 = p3.getX(); // x3 vale 3.0
```

ÍNDICE

5. Tipo Comparable

- El tipo *Comparable* define un orden sobre los objetos de un tipo creado por el usuario, al que llamaremos orden natural.
- Java ya proporciona un orden natural para los tipos envoltura como Integer y Double o el tipo inmutable String permitiendo que, por ejemplo, se puedan ordenar cuando forman parte de una lista.
- El tipo *Comparable* se compone de un único método, de nombre *compareTo*. El tipo *Comparable* usa un tipo genérico (T) y establece el orden natural sobre los objetos de un tipo dado.

```
public interface Comparable<T> {
      int compareTo(T o);
}
```

5. Tipo Comparable - Ejemplo

- El método *compareTo* compara dos objetos *o1* y *o2* y devuelve un entero que es:
  - Negativo si *o1* es menor que *o2*
  - Cero si *o1* es igual a *o2*
  - Positivo si *o1* es mayor que *o2*

```
public int compareTo(Estudiante e) {
    int r;
    if (p == null) {
              throw new NullPointerException();
    }
    r = getNombre().compareTo(p.getNombre());
    if (r == 0) {
              r = getEdad().compareTo(p. getEdad());
              if (r == 0) {
                        r = getDNI().compareTo(p.getDNI());
              }
    }
    return r;
}
```

```
public class Estudiante implements
Comparable<Estudiante> {
                  ...
}
```

ÍNDICE

6. Restricciones

- En el esquema de diseño de tipos aparece la palabra 'Restricciones' junto a cada propiedad.
- Pueden existir ciertas restricciones sobre los valores que pueden tomar las propiedades. Cuando se crea un objeto, hay que comprobar que se cumplen todas las restricciones que existan sobre sus propiedades:
  - La duración de una canción no puede ser negativa.
  - La fecha de llegada de un vuelo no puede ser anterior a la fecha de salida.
  - El nombre de una persona no puede estar vacío.
- Cuando se crea un objeto, hay que comprobar que se cumplen todas las restricciones que existan sobre sus propiedades.

6. Excepciones

- Evento que ocurre durante la ejecución de un programa, y que indica una situación normal o anormal que hay que gestionar.
- Otro mecanismo de control. Es decir, son un instrumento para romper y gestionar el orden en que se evalúan las sentencias de un programa. Por ejemplo:
  - Una división por cero.
  - Acceso a un fichero no disponible en el disco.
- Para tratarlas hay dos opciones:
  - Lanzarlas
  - Capturarlas

6. Excepciones - Throw


```
if (condicion_de_lanzamiento) {
        throw new tipo_excepcion("Texto informativo");
}
```

```
public void setNombre(String nombre) {
   if (nombre.equals("")) {
          throw new IllegalArgumentException("El nombre no puede estar vacío");
   }
   this.nombre = nombre;
}
```

6. Excepciones – Try/Catch

- En el ejemplo anterior, el programa finaliza lanzando una excepción y mostrando el correspondiente mensaje de error en la consola.
- Podemos gestionar la excepción de forma que el programa finalice de una forma más controlada. Para ello, en lugar de lanzar la excepción, la podemos **capturar**, es decir, detectar que se ha producido, y tomar las acciones necesarias para que el programa finalice de la forma más adecuada posible.
- Para ello usamos la cláusula *try/catch*. Veamos cómo sería en el ejemplo anterior.

```
try {
   Estudiante e = new Estudiante(“Antonio", 20, “12345678A");
   System.out.println(p);
} catch (IllegalArgumentException e) {
        System.out.println("Excepción capturada: nombre vacío");
}
System.out.println("Programa finalizado");
```

ÍNDICE

7. Constructor a partir de String

- Con el método *toString()* convertimos un objeto en su representación como cadena de caracteres. Muchos tipos necesitan la operación inversa, es decir, construir un objeto a partir de la información contenida en una cadena de caracteres.
- Para poder implementar esta funcionalidad necesitamos que el tipo correspondiente tenga un constructor que tome como único parámetro una cadena de caracteres.
- Con esta funcionalidad pretendemos poder construir, por ejemplo, objetos de tipo Punto a partir de cadenas de la forma "(5.3, -2.1)".

```
Integer a = new Integer("+35");
Double b = new Double("-4.57");
```

7. Constructor a partir de String


```
public Estudiante(String s) {
   String[] sp = s.split(",");
   if (sp.length != 4) {
          throw new IllegalArgumentException("Cadena con formato no válido");
   }
   String nombre = sp[1].trim();
   this.nombre = nombre;
   this.edad = Integer.valueOf(sp[1].trim());
   this.dni = sp[2].trim();
   this.grupo = Grupo.valueOf(sp[3].trim());
   this.nota1 = 0.0;
   this.nota2 = 0.0;
   this.nota3 = 0.0;

}
```

ÍNDICE

8. Records

- A partir de Java 14
- Tipo inmutable para almacenar datos simples como consultas a BBDD, servicio, o fichero de texto.
- Al ser inmutables no tienen métodos modificadores (setters)

```
public record Estudiante(String nombre, Integer edad, String dni, Grupo grupo,
Double nota1, Double nota2, Double nota3){    }
```

```
Estudiante e = new Estudiante("Antonio", 20, "12345678A", Grupo.A1, 4.1, 9.2, 8.75);
System.out.println(e);
```

8. Records


```
public record Estudiante(String nombre, Integer edad, String dni,
Grupo grupo, Double nota1, Double nota2, Double nota3){
	public Estudiante {
		checkNombre(nombre);
	}

	private void checkNombre(String nombre) {
		if (nombre.equals("")) {
			throw new IllegalArgumentException("El nombre no
puede estar vacío");
		}
	}
}
```

ÍNDICE

9. Clases de utilidad

- Son clases que contienen una colección de métodos que realizan unas tareas útiles en un determinado dominio. Estas clases normalmente no tienen atributos, sino solo un conjunto de métodos **static**.
- Un ejemplo de clase de utilidad es la clase **Math** de la API de Java. Esta clase contiene métodos que realizan operaciones matemáticas, como la raíz cuadrada, el seno y el coseno, el valor absoluto, etc.
- También podemos definir nuestras propias clases de utilidad. Por ejemplo, la clase de utili0dad Checkers contiene métodos para chequear los valores de los atributos de un objeto

9. Clases de utilidad


```
public class Checkers {
         public static void check(String textoRestriccion, Boolean condicion) {
            if (!condicion) {
               throw new IllegalArgumentException(
            Thread.currentThread().getStackTrace()[2].getClassName()
            + "." +
            Thread.currentThread().getStackTrace()[2].getMethodName()
            + ": " + textoRestriccion);
            }
         }

         public static void checkNoNull(Object... parametros) {
            for (int i = 0; i < parametros.length; i++) {
         if (parametros[i] == null) {
            throw new IllegalArgumentException(Thread.currentThread().getStackTrace()[2].getClassName()
         + "." + Thread.currentThread().getStackTrace()[2].getMethodName() + ": el parámetro " + (i + 1) + " es nulo");
         }
            }
         }
}
```