Sesión 20: Herencia 1/2

Programación 2

Ángel Herranz

2020-2021

Universidad Politécnica de Madrid

En capítulos anteriores

- 👍 Tema 1: Intro a POO
- 👍 Tema 2: Clases y Objetos y TADs y módulos
- 👍 Tema 3: Colecciones con *arrays*
- 👍 Tema 4: Cadenas simplemente enlazadas
- 🕒 Tema 5: TAD Listas
- 🕒 Tema 7: Herencia[1] y Polimorfismo
  - ***Ad hoc***: mencionaremos sobrecarga
  - **Genéricos**: polimorfismo paramétrico

[1] Sólo de interfaz.

En el capítulo de hoy

- Tema 7: Herencia y Polimorfismo
  - **Subtipado**: ¡empezamos!


“Un gran poder. . .”
Stan Lee

Supongamos que nos regalan listas[2]

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

[2] Són listas de `Integer` y sólo se presenta el API, ¿para qué quieres más?

Nos piden implementar conjuntos (c)

```
public class Set<T> {

  public Set() {

  }
  public void add(T e) {



  }
  public int size() {

  }
  public boolean includes(T e) {

  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Nos piden implementar conjuntos (c)

```
public class Set<T> {
  private List<T> data;
  public Set() {

  }
  public void add(T e) {


  }
  public int size() {

  }
  public boolean includes(T e) {

  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Composición

(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set<T> {
  private List<T> data;
  public Set() {
    data = new List<T>();
  }
  public void add(T e) {



  }
  public int size() {


  }
  public boolean includes(T e) {


  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Composición

(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set<T> {
  private List<T> data;
  public Set() {
    data = new List<T>();
  }
  public void add(T e) {



  }
  public int size() {
    return data.size();
  }
  public boolean includes(T e) {


  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Composición

(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set<T> {
  private List<T> data;
  public Set() {
    data = new List<T>();
  }
  public void add(T e) {



  }
  public int size() {
    return data.size();
  }
  public boolean includes(T e) {
    return data.indexOf(e)!=-1;
  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Composición

(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set<T> {
  private List<T> data;
  public Set() {
    data = new List<T>();
  }
  public void add(T e) {
    if (!this.includes(e))
      data.add(0, e);
  }
  public int size() {
    return data.size();
  }
  public boolean includes(T e) {
    return data.indexOf(e)!=-1;
  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Composición

(*regla has-a*)

Nos piden implementar conjuntos (h)

```
public class Set<T> extends List<T>
{
  public Set() {
  }
  public void add(T e) {


  }
  public boolean includes(T e) {

  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Herencia

(*regla is-a*)

Nos piden implementar conjuntos (h)

```
public class Set<T> extends List<T>
{
  public Set() {
  }
  public void add(T e) {
    if (!this.includes(e))
      this.add(0, e);
  }
  public boolean includes(T e) {
    return this.indexOf(e)!=-1);
  }
}
```

```
public class List<T> {
  public List();
  public void add(int i, T e);
  public T get(int i);
  public int size();
  public void remove(int i);
  public void remove(T e);
  public int indexOf(T e);
}
```

Herencia

(*regla is-a*)

Herencia

- Concepto fundamental en OO
- En Java, para empezar:

```
class A extends B {...}
```

[3] Atributos y métodos.

Herencia

- Concepto fundamental en OO
- En Java, para empezar:

`class` *A* **`extends`** *B* `{...}`

Las instancias de la clase *A* tienen todas las propiedades[3] declaradas en *B*

[3] Atributos y métodos.

Herencia

- Concepto fundamental en OO
- En Java, para empezar:

`class` *A* **`extends`** *B* `{...}`

Las instancias de la clase *A* tienen todas las propiedades[3] declaradas en *B*

- Decimos que una clase *A* hereda de otra clase *B*
- También decimos que *A* es subclase de *B*
- También decimos que *B* es superclase de *A*

[3] Atributos y métodos.

💬 Preguntas

- ¿Dónde está el método `size` en (h)?
- ¿Qué cosas puedo hacer con un conjunto en (c)?
- ¿Y en (h)?
- ¿Es mejor (c) o (h)?

Ejemplo “tonto”

- Clase `Mamífero`: `comunicar()`

  Todos los mamíferos se comunican

- Clase `Perro`: `comunicar()` y `ladrar()`

  Los perros además de comunicar, ladran

- Clase `Gato`: `comunicar()` y `maullar()`

  Los perros además de comunicar, maullan

- Programa principal que *juegue* con objetos de dichas clases[4]

[4] Ej. ¿cómo se comunican los mamíferos? ¿puedo hacer que un perro se comunique de una forma especial? ¿y un gato? ¿puedo declarar una variable de tipo mamífero y poner una referencia a un perro? ¿y al revés?

`geometria` con herencia

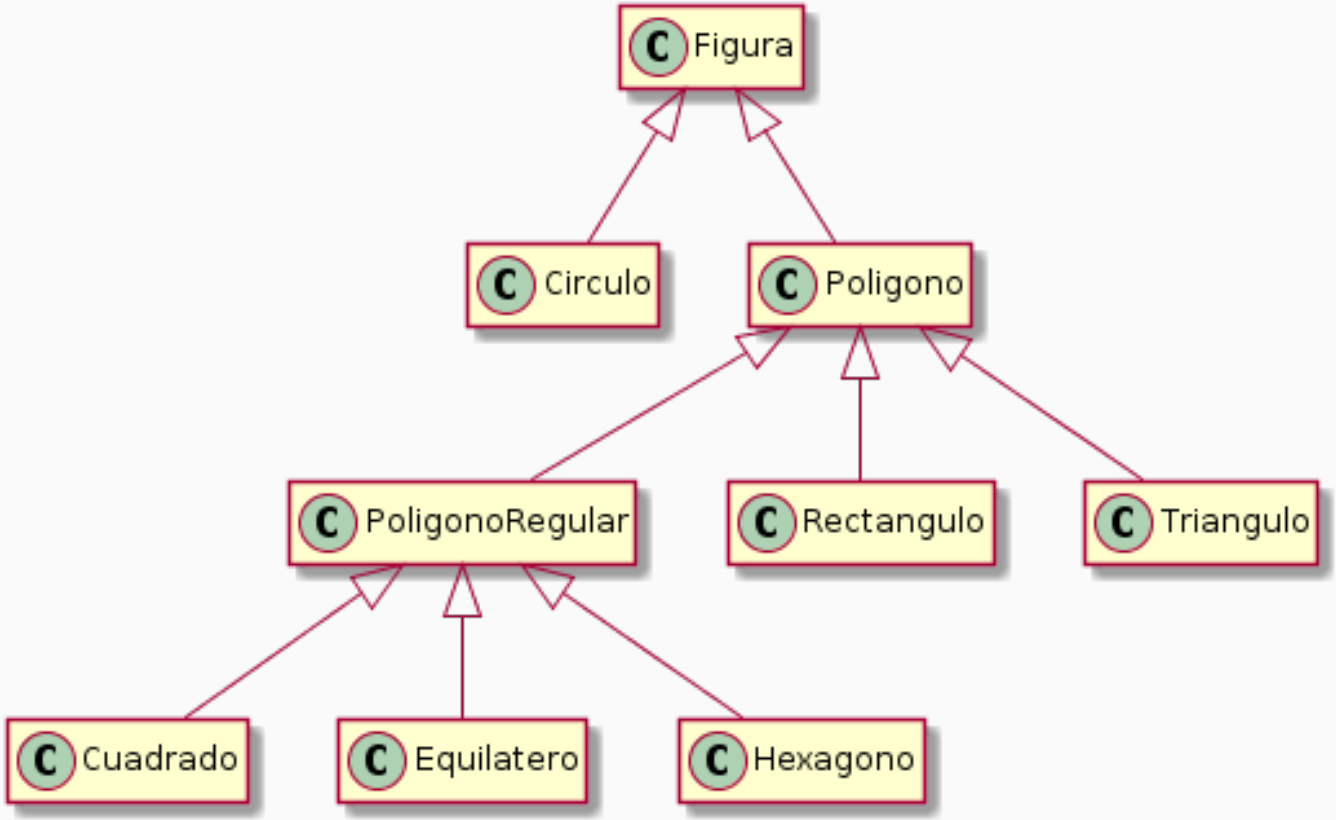
- `Figura`
- `Circulo`
- `Poligono`
- `PoligonoRegular`
- `Rectangulo`
- `Cuadrado`
- `Triangulo`
- `Equilatero`
- `Hexagono`
- Y un programa principal para probar

¡Aprendiendo!

- Las siguientes transparencias contienen el resultado de un diseño colaborativo en clase[5]
- La idea es que fueran surgiendo necesidades y soluciones a medida que se avanzaba
- La clase comenzó con una "clasificación" de las clases por herencia
- Las clases se fueron implementando de arriba a abajo siguiendo el árbol de herencia

[5] Curso 2018-2019.

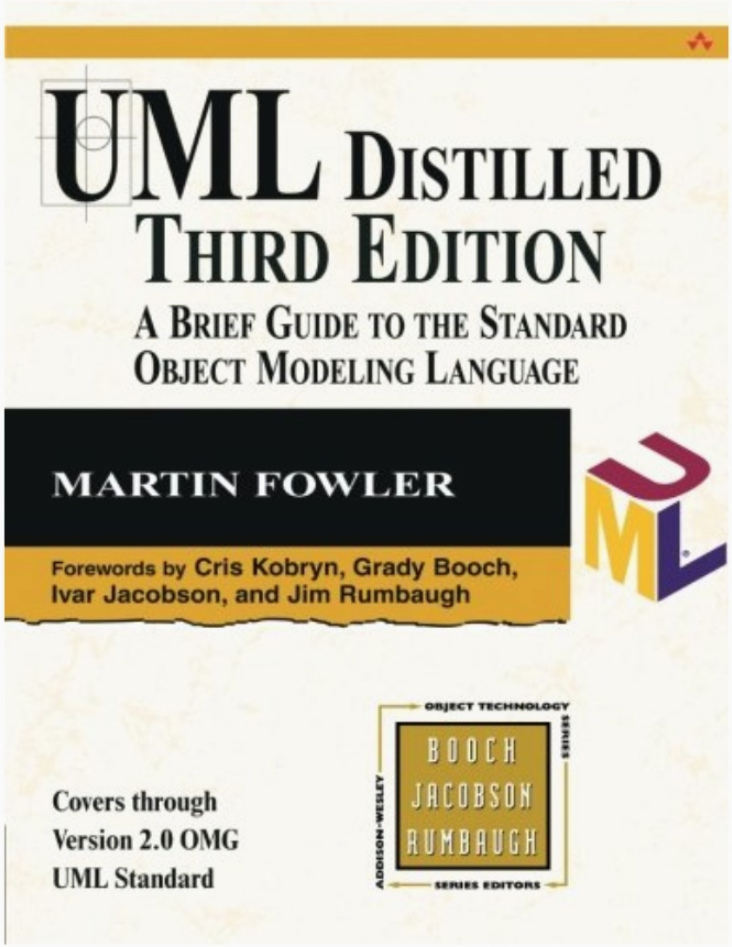
Jerarquía de herencia[6]



[6] Java no permite herencia múltiple: por ejemplo, no es posible hacer

```
public class Equilatero extends Triangulo, PoligonoRegular...
```

Unified Modelling Language (UML)



- Lenguaje gráfico de diagramas (muy orientado a objetos)
- Podemos representar modelos del sistema a construir
- Uno de los diagramas nos permite representar clases

Empezamos por *el top*: `Figura`

- Damos por supuesta la existencia de la clase `Punto2D` (`distancia`, `equals`, etc.)
- Todas las figuras tienen un centro
- Todas las figuras tienen un área

Deberías estar programando. . . [a]

[a] no mirar hasta haberlo intentado por ti mismo

Empezamos por *el top*: `Figura`

Deberías estár programando...

Empezamos por *el top*: `Figura`

Deberías estár programando...

```
package geometria;
public class Figura {
  private Punto2D centro;
  public Punto2D centro() {
    return this.centro;
  }
  public double area() {
    return 0; // ¿Qué otra cosa podemos hacer?
  }
}
```

Primera subclase: `Circulo`

- Todos los círculos tienen centro (¡son figuras!)
- Todos los círculos tienen área (¡son figuras!)
- Todos los círculos, además, tienen un radio

Primera subclase: `Circulo`

Deberías estár programando...

Primera subclase: `Circulo`

Deberías estár programando...

```
package geometria;
public class Circulo extends Figura {
  private double radio;
  public Circulo(Punto2D c, double r) {
    centro = c;
    radio = r;
  }
}
```

Probemos ya: `GeoTest`

Deberías estár programando...

Probemos ya: `GeoTest`

Deberías estár programando...

```
import geometria.*;
public class GeoTest {
  public static void main(String[] argv) {
    Circulo c = new Circulo(new Punto2D(0,0), 2.0);
    assert c.centro().equals(new Punto2D(0,0));
    assert Math.abs(c.area() - 4 * Math.PI) < 0.001;
  }
}
```

```
javac -d lib -cp .:lib -sourcepath .:src src/GeoTest.java
java -ea -cp lib GeoTest
```

¿Problemillas?

- ¿Algún problema con el constructor de `Circulo`?
- ¿Quizás con `centro`?

 ¡Intenta resolverlo *como sea*!

¿Problemillas?

- ¿No pasa los tests?
- ¿Quizás el área?

Deberías de ser capaz de resolverlo

Sobreescribe el método `área`

Modificador de visibilidad: `protected`

Para que una subclase pueda usar una propiedad de alguna de su superclases, la propiedad debe etiquetarse al menos con el modificador ***`protected`***

Clases *abstractas*

La clase `Figura` *no sabe calcular el área de cualquier figura, es demasiado abstracta, cuando esto sucede podemos marcar el método con el modificador* **`abstract`**

Figura: clase abstracta[7] + protected[8]

```
package geometria;

public abstract class Figura {
  protected Punto2D centro;

  public abstract double area();

  public Punto2D centro() {
    return centro;
  }
}
```

[7] Métodos sin implementar: serán implementados en las subclases

[8] Visibilidad en la clases y en sus subclases

Pruebas con `Figura`

Deberías estár programando...

Pruebas con `Figura`

Deberías estár programando...

```
Figura f = new Figura();
assert f.centro() == null;
```

Pruebas con `Figura`

Deberías estár programando...

```
Figura f = new Figura();
assert f.centro() == null;
```

No es posible crear, directamente,
instancias de clases abstractas

Poligono

- Todos los polígonos son figuras
- Los polígonos tienen, además, un número de lados
- Los polígonos tienen, además, un perímetro

Deberías estár programando. . .

`Poligono`: aún es demasiado abstracta[9]

```
package geometria;

public abstract class Poligono extends Figura
{
  protected int nLados;

  public abstract double perimetro();

  public int nLados() {
    return nLados;
  }
}
```

[9] Aún no se puede programar `area()` y se añade `perimero()` que tampoco se sabe cómo implementar

PoligonoRegular

- Todos los polígonos regulares son polígonos
- Los lados de un polígono regular son iguales (longitud del lado)
- Calcular el área y el perímetro de un polígono regular es "fácil": ¡ya no es abstracta!

Deberías estár programando. . .

PoligonoRegular i

```
package geometria;

public class PoligonoRegular extends Poligono {
  protected double longLado;

  public PoligonoRegular(Punto2D centro,
                         int nLados,
                         double longLado) {
    this.centro = centro;
    this.nLados = nLados;
    this.longLado = longLado;
  }
```

PoligonoRegular ii

```
  public double perimetro() {
    return longLado * nLados;
  }

  public double lado() {
    return longLado;
  }

  public double area() {
    double apotema = longLado / (2 * Math.tan(Math.PI/nLados));
    return nLados * apotema * longLado / 2;
  }
}
```

Deberías estár programando. . .

Hexagono: super[10] + sobreescritura[11]

```
package geometria;

public class Hexagono extends PoligonoRegular
{
  public Hexagono(Punto2D centro,
                  double longLado) {
    super(centro, 6, longLado);
  }

  public double area() {
    return 3 * Math.sqrt(3) * longLado * longLado / 2;
  }
}
```

[10] Reusando el constructor del *padre*

[11] *Overriding*: sobreescribimos el método `area()` con mayor "eficiencia"