

Sesión 19: Herencia 1/2

Programación 2

Ángel Herranz

2021-2022

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: Herencia¹ y Polimorfismo
 - ***Ad hoc***: mencionaremos **sobrecarga**
 - **Genéricos**: polimorfismo **paramétrico**

¹Sólo de interfaz.

En el capítulo de hoy

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

“Un gran poder...”

Stan Lee



Supongamos que nos regalan listas²

```
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) { ... }  
}
```

²Sólo se presenta el API.

Nos piden implementar conjuntos (c)

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 (c)
(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 (c)
(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 (c)
(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 (c)
(*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 (c)
(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 (h)
(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 (h)
(regla *is-a*)

Herencia

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

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

³Atributos 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*

³Atributos 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*

³Atributos 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” i

- Clase Mamífero: `comunicar()`
Todos los mamíferos se comunican
"Hola soy un mamífero"
- Clase Perro: `comunicar()` y `ladrar()`
Los perros además de comunicar, ladran
"Guau!"
- Clase Gato: `comunicar()` y `maullar()`
Los gatos además de comunicar, maullan
"Miau!"
- Programa principal que *juegue* con objetos de dichas clases

Ejemplo “tonto” ii

- ¿Puedo guardar un objeto de la clase Perro en una variable m de tipo Mamifero?
- ¿Y al revés?
- ¿Puedo hacer `m.ladRAR()`?
- ¿Puedo cambiar (*sobreescibir*) el método comunicar en Perro para que diga "Hola soy un perro"?
- ¿Qué pasa cuando haga `p.comunicar()`? ¿Y `m.comunicar()`?
- ¿Otras preguntas?

geometria con herencia

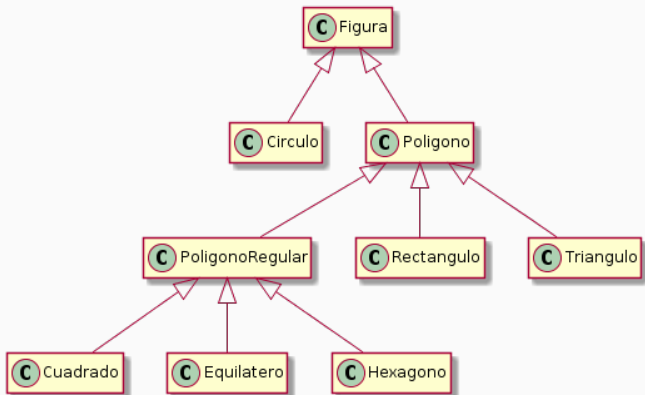
- Figura
- Circulo
- Poligono
- PoligonoRegular
- Rectangulo
- Cuadrado
- Triangulo
- Equilatero
- Hexagono
- Y un programa principal para probar

¡Aprendiendo!

- Las siguientes transparencias contienen el resultado de un diseño colaborativo en clase⁴
- 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

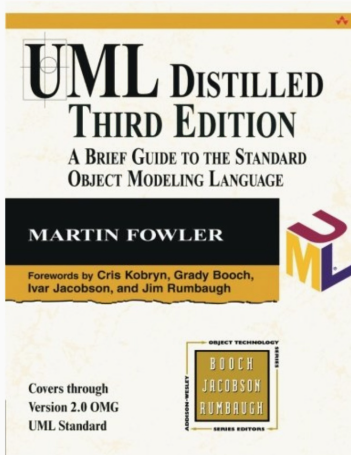
⁴Curso 2018-2019.

Jerarquía de herencia⁵



⁵Java **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

^ano mirar hasta haberlo intentado por ti mismo



Empezamos por *el top*: Figura

Deberías estar programando...



Empezamos por *el top*: Figura

Deberías estar 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: **Círculo**

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

Primera subclase: **Círculo**

Deberías estar programando...

Primera subclase: **Círculo**

Deberías estar programando...

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



Probemos ya: GeoTest

Deberías estar programando...



Problemos ya: GeoTest

Deberías estar 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 abstracta⁶ + **protected**⁷

```
package geometria;  
  
public abstract class Figura {  
    protected Punto2D centro;  
  
    public abstract double area();  
  
    public Punto2D centro() {  
        return centro;  
    }  
}
```

⁶Métodos sin implementar: serán implementados en las subclases

⁷Visibilidad en la clases y en sus subclases

Pruebas con **Figura**

Deberías estar programando. . .

Pruebas con Figura

Deberías estar programando...

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

Pruebas con Figura

Deberías estar programando...

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

No es posible crear, **directamente**,
instancias de clases abstractas

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

Deberías estar programando...

Poligono: aún es demasiado abstracta⁸

```
package geometria;

public abstract class Poligono extends Figura
{
    protected int nLados;

    public abstract double perimetro();

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

⁸Aún no se puede programar `area()` y se añade `perimetro()` que tampoco se sabe cómo implementar

PolígonoRegular

- 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 estar 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 estar programando...

Hexagono: **super**⁹ + sobrescritura¹⁰

```
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;
    }
}
```

⁹Reusando el constructor del *padre*

¹⁰*Overriding*: sobrescribimos el método `area()` con mayor “eficiencia”