Programación 2

Herencia y Clases Abstractas

Profesores:
Ismael Figueroa - ifigueroap@gmail.com
Eduardo Godoy - eduardo.gl@gmail.com

¿Qué es la herencia?

- Es un mecanismo para *definir nuevas clases en base a otras ya existentes*
- A la clase existente se le llama: clase base, superclase, o clase padre. A la clase nueva se le llama: clase extendida, subclase, o clase hija
- La subclase *hereda* o *adquiere* todos los atributos y métodos de la superclase
- La subclase puede usar el constructor de la super clase mediante la palabra clave `super`

Ejemplo de Herencia

```
public class Rectangulo {
  Integer largo, ancho;

  public Rectangulo(Integer l,
                    Integer a) {
   largo = l;
   ancho = a;
  }

  public Integer area() {
    return largo*ancho;
  }

  public Integer perimetro() {
    return 2*(ancho + largo);
  }
}
```

```
public class Cuadrado
extends Rectangulo {
  /* Hereda: largo, ancho,
  area y perimetro */
  public Cuadrado(Integer lado) {
    super(lado, lado);
  }
}

public class Main {
  public static
  void main(String[] args) {
    Rectangulo r = new Rectangulo(4,2);
    Cuadrado c   = new Cuadrado(5);
    /* ... */
  }
}
```

Especialización de la subclase

- Se dice que la superclase *es más general* que la subclase
- Al reves, la subclase *es una especialización* de la superclase
- La herencia se usa para codificar relaciones *"es un"*. Por ejemplo: *un Cuadrado* ***es un*** *Rectángulo*
- Al ser más específica, una subclase puede agregar nueva información mediante *nuevos atributos* y *nuevos métodos*
- Una subclase también puede ser más específica al ***sobre-escribir*** métodos de la superclase. Se usa la anotación `@Override` para que el compilador sepa nuestra intención!

Especialización: nuevos atributos

```
public class Rectangulo {
  Integer largo, ancho;

  public Rectangulo(Integer l,
                    Integer a) {
   largo = l;
   ancho = a;
  }

  public Integer area() {
    return largo*ancho;
  }

  public Integer perimetro() {
    return 2*(ancho + largo);
  }
}
```

```
public class RectColor
extends Rectangulo {
  String color;

  public RectColor(Integer l,
                   Integer a,
                   String color) {
      super(l, a);
      this.color = color;
  }
}

public class Main {
  public static
  void main(String[] args) {
    RectColor rc;
    rc = new RectColor("Rojo", 4, 2);
    /* ... */
  }
}
```

Especialización: nuevos métodos

```
public class Rectangulo {
  Integer largo, ancho;

  public Rectangulo(Integer l,
                    Integer a) {
   largo = l;
   ancho = a;
  }

  public Integer area() {
    return largo*ancho;
  }

  public Integer perimetro() {
    return 2*(ancho + largo);
  }
}
```

```
public class RectColor
extends Rectangulo {
  String color;
  /* ... constructor ...*/

  public void setColor(String c) {
    color = c;
  }
}

public class Main {
  public static
  void main(String[] args) {
    RectColor rc;
    rc = new RectColor("Rojo", 4, 2);
    rc.setColor("Verde");
    /* ... */
  }
}
```

Especialización: **sobreescribir** métodos

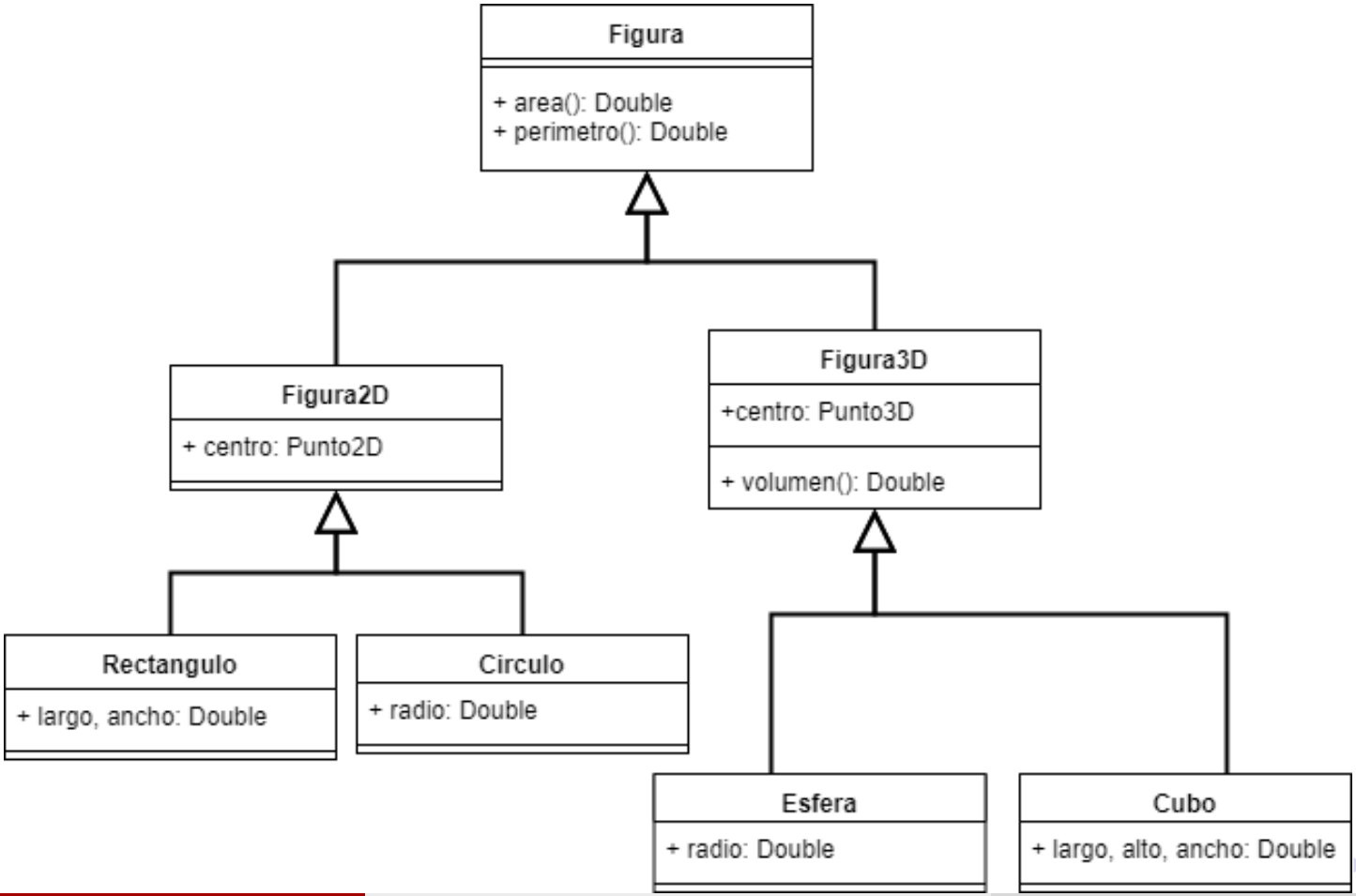
Cuando una subclase sobreescribe un método de una superclase estamos asociando un comportamiento más específico a una misma acción que en la superclase es más general. Lo clásico que hemos visto hasta ahora es sobreescribir el método `toString()`, que viene transitivamente desde la superclase `Object`.

```
public class Rectangulo {
  Integer largo, ancho;
  /* ... */

  @Override // no olvidar el override!
  public String toString() {
    return String.format(
        "Un hermoso rectangulo de largo %d y %ancho %d", largo, ancho);
  }
}
```

Ejemplo: Figuras Geométricas

Podemos ir más allá de los rectángulos a una jerarquía de clases más completa, considerando una clase `Figura`, especializada en `Figura2D`, y `Figura3D`, y figuras concretas en cada caso....



Métodos destinados a ser sobreescritos

Para la clase `Figura`, es imposible saber cuál es una buena implementación para `area` y `perimetro`, ya que dependen de información específica de cada figura...

```
public class Figura {
  public Double area() {
    /* ??? */
  }

  public Double perimetro() {
    /* ??? */
  }
}
```

Métodos destinados a ser sobreescritos

Una opción es arrojar una excepción:

```
public class Figura {
  public Double area() {
   throw new UnsupportedOperationException("Debe refinar en subclases");
 }

 /* ... */
}
```

Pero esto es "trampa", porque en realidad queremos decir que todas las figuras deben implementar, según su contexto, los métodos `area` y `perimetro`

Métodos Abstractos

- Son métodos en los que solo se declara su tipo de retorno, nombre, y parámetros, ***pero que no tienen implementación***.
- Se definen cuando se requiere que una subclase complete ese método con una implementación concreta y específica
- Si una clase incluye un método abstracto, entonces obligatoriamente ***la clase también debe ser abstracta***

Clases Abstractas

- Son clases diseñadas para *nunca ser instanciadas*!!
- Su propósito es funcionar como superclase en una jerarquía de herencia
- Una clase abstracta puede declarar "piezas faltantes" que **deben** ser implementadas por las subclases para convertirse en una *clase concreta*
- Las piezas faltantes pueden ser atributos y principalmente métodos abstractos

Una Figura abstracta

```
public abstract class Figura {
  public abstract Double area();
  public abstract Double perimetro();
}

public class Punto2D {
  Integer x, y;
  public Punto2D(Integer x, Integer y) {
    this.x = x;
    this.y = y;
  }
}

public class Figura2D extends Figura {
  public Punto2D centro;
  public Figura2D(Punto2D centro) {
    this.centro = centro;
  }
}
```

Un Rectangulo concreto

```
public class Rectangulo extends Figura2D {
  Integer largo, ancho;
  public Rectangulo(Punto2D centro, Integer largo, Integer ancho) {
    super(centro);
    this.largo = largo;
    this.ancho = ancho;
  }

  @Override
  public Double area() {
    return largo * ancho;
  }

  @Override
  public Double perimetro() {
    return 2*(largo + ancho);
  }
}
```

Variables public

Observe el código de Figura2D:

```
public class Figura2D extends Figura {
  public Punto2D centro;
  public Figura2D(Punto2D centro) {
    this.centro = centro;
  }
}
```

El atributo es public por lo que cualquier usuario de nuestra librería de figuras podría cambiar la posición de una figura. Esto ciertamente no es deseable, pero ...

Variables private

Si hacemos que el centro sea private, tendremos un error en:

```
public Rectangulo(Punto2D centro, Integer largo, Integer ancho) {
  this.centro;
  this.largo = largo;
  this.ancho = ancho;
}
```

El error nos dice que ***no podemos acceder al atributo*** *centro*!! Pero se suponía que por herencia obteníamos todo...
En Java las variables private están ocultas en las subclases, y sólo se pueden acceder mediante los métodos de la superclase que los manipulan...

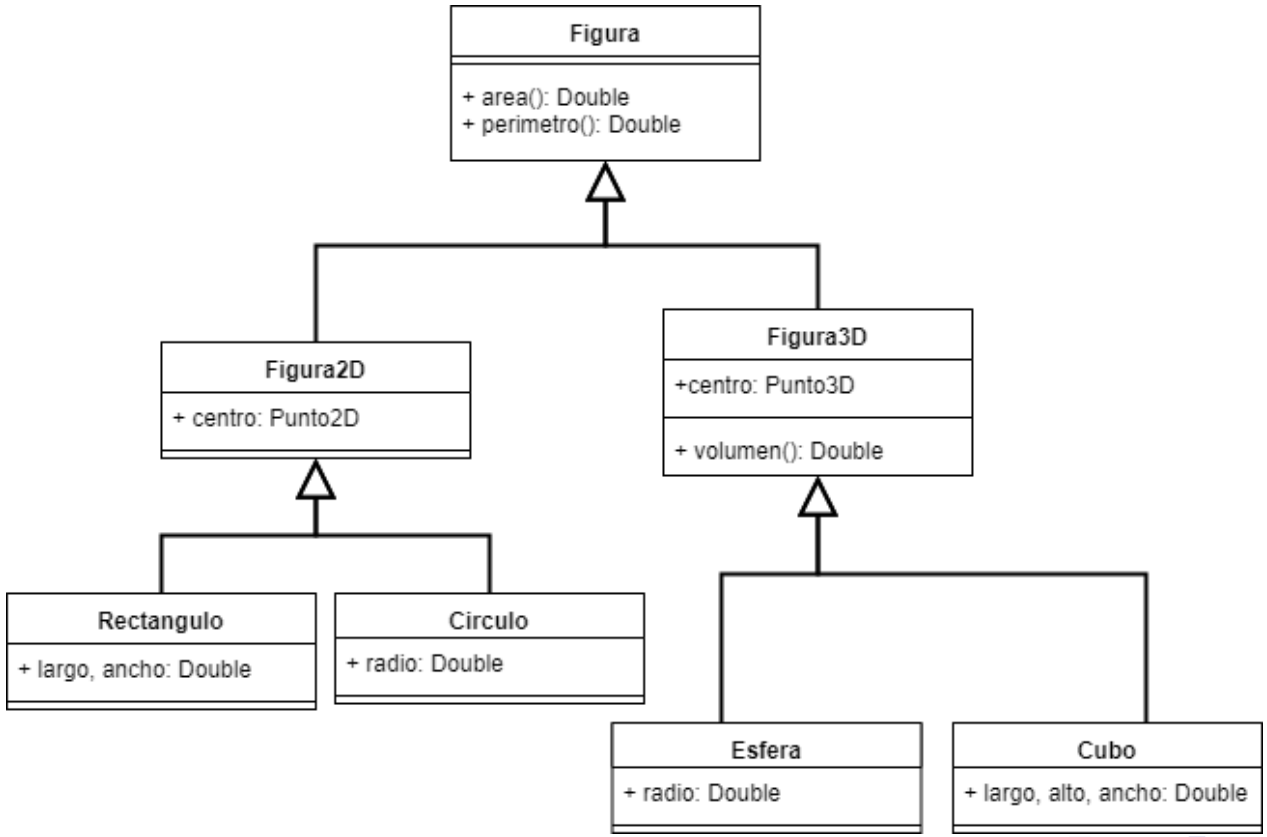
Nuevo acceso: variables `protected`

Java incorpora un nuevo modificador de acceso: `protected`

- Permite que el campo sea visible en las subclases
- Mantiene el nivel de acceso dentro de la misma clase y el mismo paquete
- Para el resto del mundo, es como un atributo privado
- Ver `docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

Codificación Figuras Geométricas

Codifique el modelo de clases que representa las figuras geométricas. Escriba un programa principal que construya una instancia de cada clase concreta, y donde se invoquen todos sus métodos. Use clases y métodos abstractos según corresponda. Todos los atributos de clase deben tener acceso `protected`.



Preguntas

Preguntas?