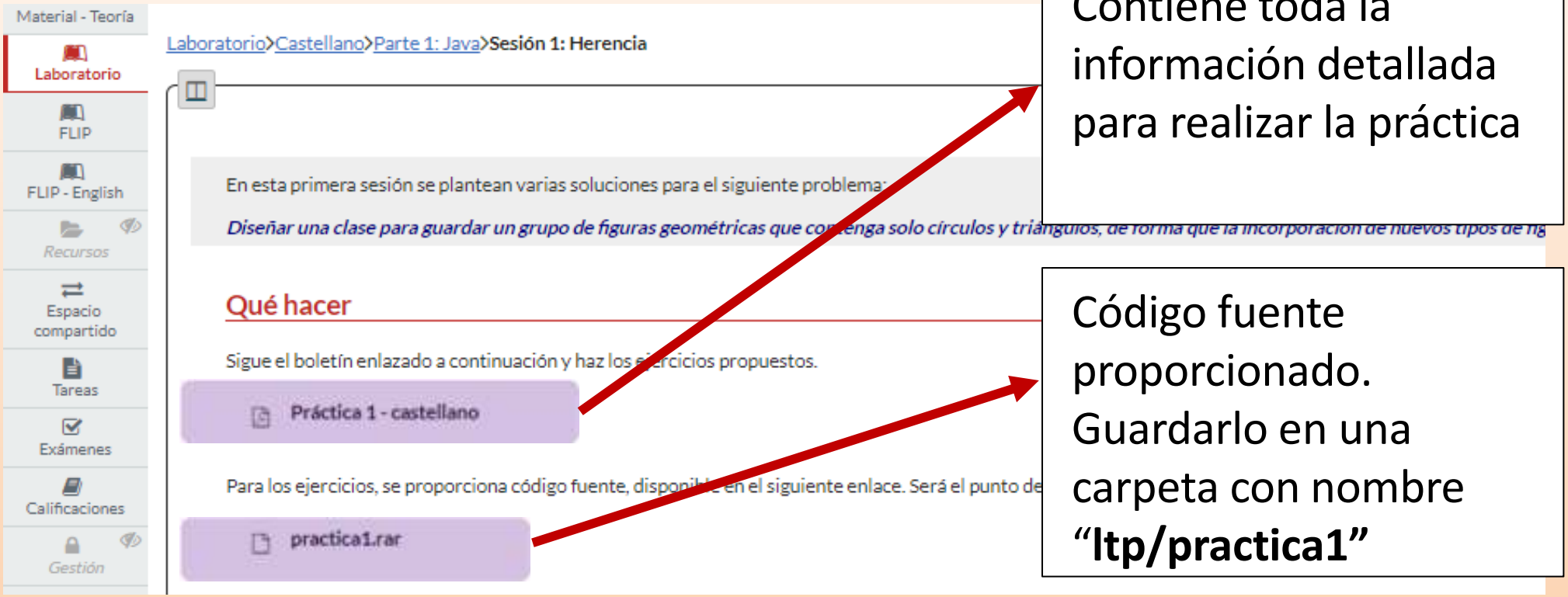
Prácticas de laboratorio de LTP
(Parte I : Java)

Práctica 1: Herencia y sobrecarga

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

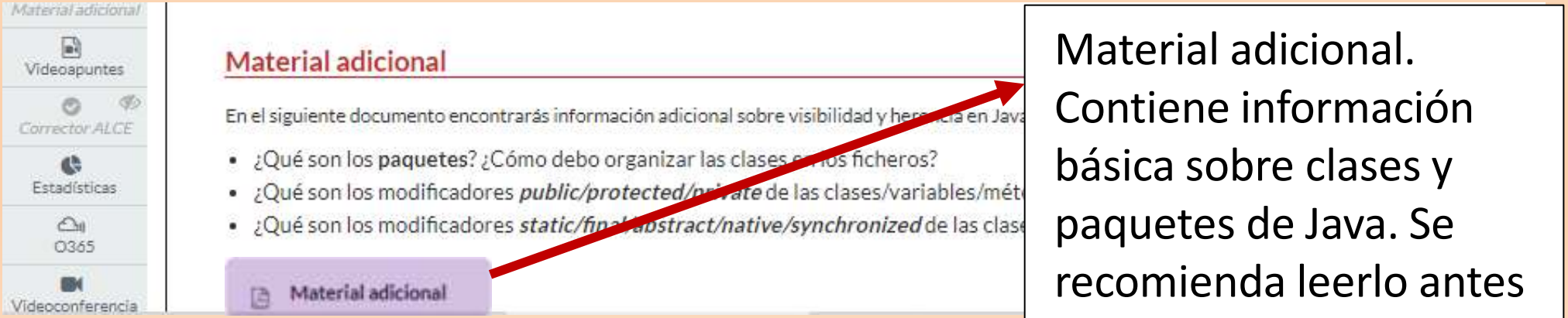
Jose Luis Pérez
jlperez@dsic.upv.es

Materiales de la práctica



Contiene toda la información detallada para realizar la práctica

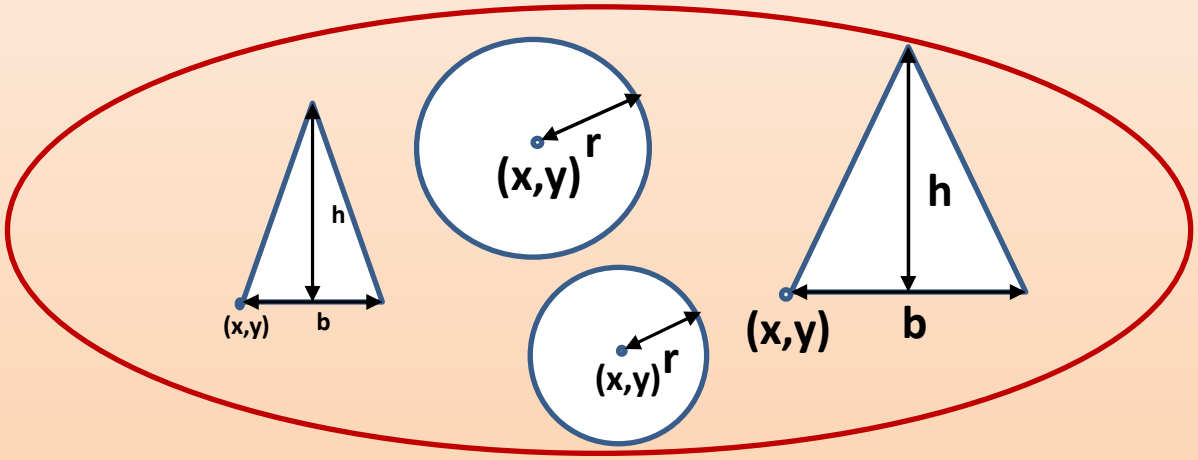
Código fuente proporcionado. Guardarlo en una carpeta con nombre "**ltp/practica1**"



Material adicional. Contiene información básica sobre clases y paquetes de Java. Se recomienda leerlo antes de la sesión.

1. Introducción : Problema a resolver

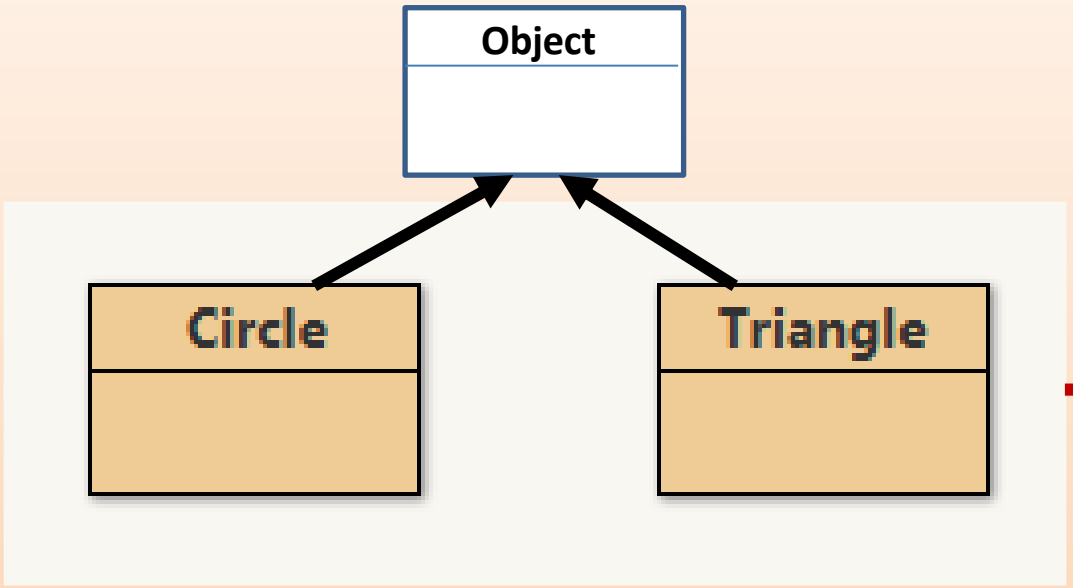
Se nos pide diseñar una **clase** para guardar información de dos tipos de figuras geométricas: **círculos** y **triángulos**.

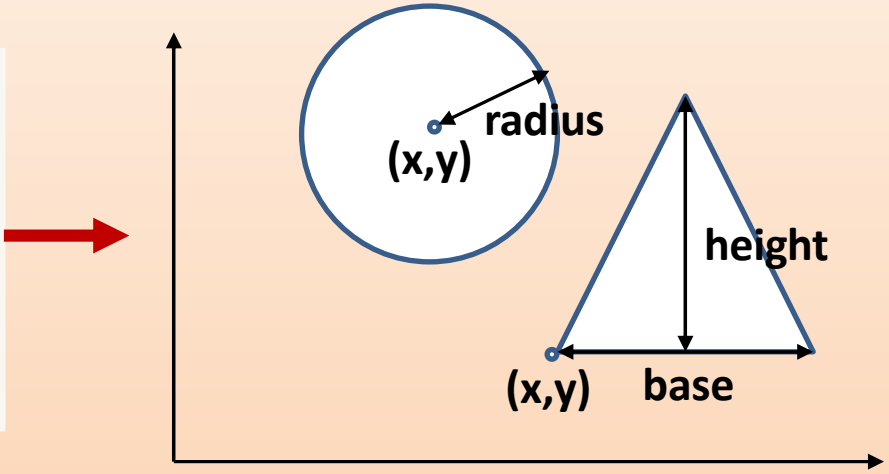


Con esta clase representaremos un conjunto de figuras, mediante sus diferentes atributos y sus posiciones en un eje bidimensional.

En esta práctica se proponen diversas opciones de implementación
El objetivo es explotar las posibilidades de la herencia de **Java** desde el punto de vista de la reutilización del código.

Planteamiento del problema **sin usar herencia**





Creamos una clase para cada tipo figura: **Circle** y **Triangle**

Cada clase tendrá sus propios atributos en relación a la figura que representa.

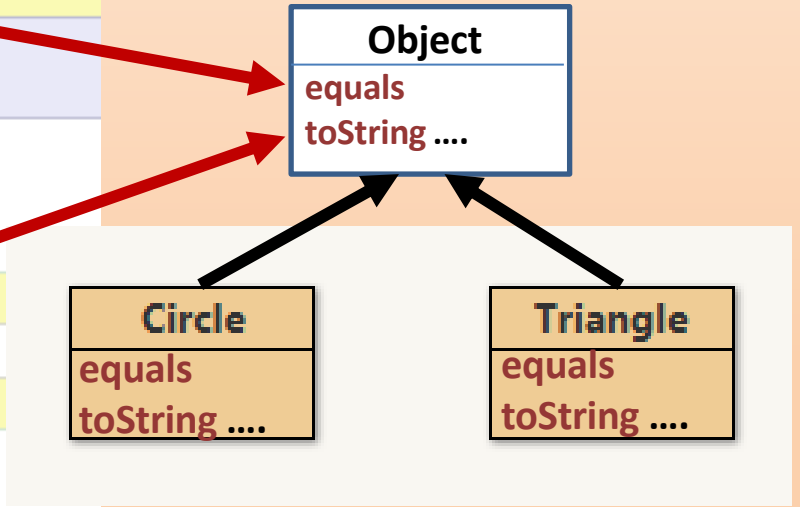
No tenemos ninguna jerarquía o clase superior que vincule esta clases (Excepto la clase **Object**)

Clase **Circle**

```
public class Circle {
 private double x, y;
 private double radius;

 public Circle(double a,double b, double c) {
      x = a; y = b; radius = c;
 }

 public boolean equals(Object o) {
   if (!(o instanceof Circle)) {
     return false; }
   Circle c = (Circle) o;
   return x == c.x && y == c.y &&
   radius == c.radius;
 }

 public String toString() {
  return "Circle:\n\t" + "Position: (" +x+ ","
         y+")\n\tRadius: " +radius;
  }

}
```

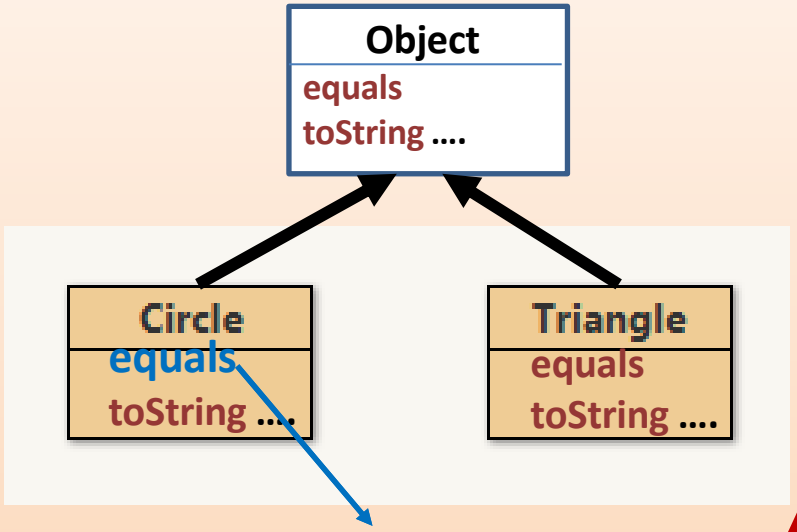
public class indica que **Circle** es una clase pública

Se define el constructor de **Circle**



Tanto **Circle** como **Triangle** van a ***sobreescribir*** los métodos **equals** y **toString** ya definidos en **Object**

Clase **Circle** : método **equals(Object)**




```
public boolean equals(Object o) {
  if (!(o instanceof Circle)) {
    return false; }
  Circle c = (Circle) o;
  return x == c.x && y == c.y &&
  radius == c.radius;
}
```

El argumento de entrada es de tipo **Object,** puede referenciar a cualquier objeto

instanceof permite averiguar si un objeto es instancia de una determinada clase. En este caso **Circle**

Si el objeto no es una instancia de **Circle,** evidentemente el resultado será **false**

Puesto que el objeto está referenciado como **Object**, no podemos acceder a los atributos de **Circle.**

La ***coerción explícita*** o ***casting*** permite disponer de una referencia a **Circle** y por tanto acceso a sus atributos.

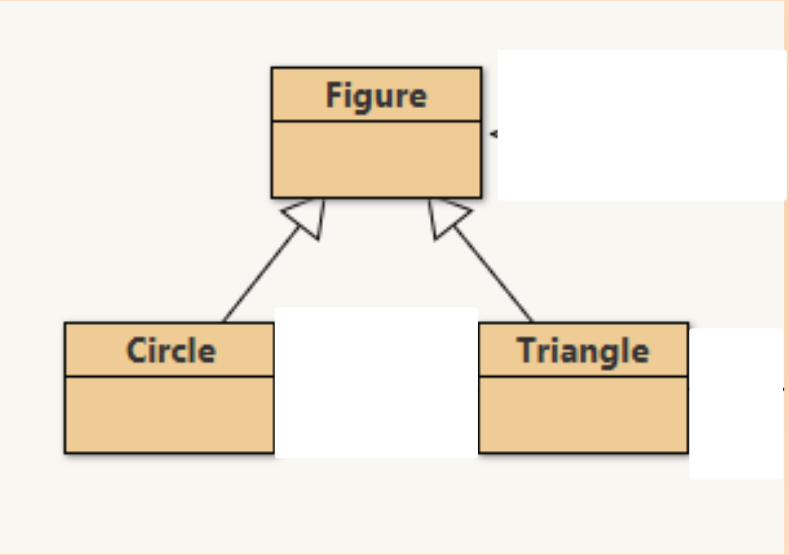
Clase **Triangle**

```
public class Triangle {
 private double x, y;
 private double base, height;

 public Triangle(double a, double b, double c, double d) {
   x = a; y = b;
   base = c; height = d;
 }

 public boolean equals(Object o) {
   if (!(o instanceof Triangle)){
   return false; }
   Triangle t = (Triangle) o;
   return x == t.x && y == t.y && base == t.base && height == t.height;
 }

 public String toString() {
   return "Triangle:\n\t" + "Position: (" +x+ "," + y+")\n\tBase: " +base+
     "\n\tHeight: " +height;
 }

}
```

2. Solución 1 : usando **herencia** y **sobrecarga**

La solución planteada sin **herencia,** tiene el inconveniente de que replica código en las clases **Circle** y **Triangle.** El problema se agravaría si decidimos ampliar el tipo de figuras.

Con el fin de mejorar la reutilización y portabilidad del código vamos a definir una nueva clase, **Figure**, que agrupe algunas de las características y comportamientos comunes de **Circle** y **Triangle.**



De esta forma creamos una jerarquía, donde la clase **Figure**, es la ***superclase*** *o clase padre*, y las clases **Circle** y **Triangle** son clases que heredan los atributos y funcionalidades definidos en la *superclase*.

Superclase : **Figure**

Figure.java

```
public class Figure {
    private double x;
    private double y;

    public Figure(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Figure)) { return false; }
        Figure f = (Figure) o;
        return x == f.x && y == f.y;
    }

    public String toString() {
        return "Position: (" + x + ", " + y + ")";
    }
}
```

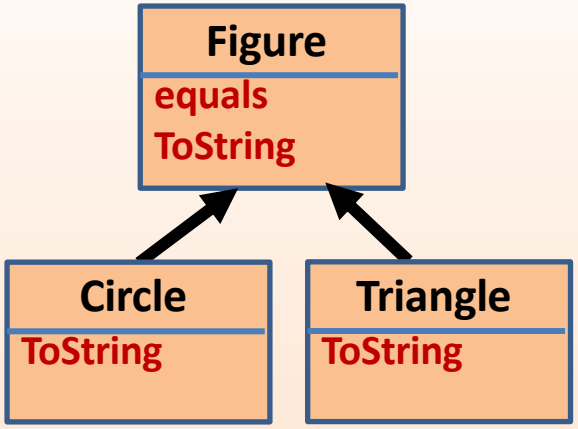
Los atributos de **Figure** serán las coordenadas que determinan la posición de la figura en el eje (x,y).

Son definidos como **private** para obligar a la reutilización del código **Figure** en las clases herederas.

This hace referencia al objeto actual, por lo que **this.x** y **this.y** son los atributos de **Figure.** En este caso es necesario utilizarlos para diferenciarlos de las variables argumento del constructor.

De nuevo sobreescribimos **equals:** Dos figuras son iguales si ocupan la misma posición en el eje (x,y).

También es necesario aplicar **instanceof** para comprobar que el argumento es una figura, y la ***coerción explícita*** para acceder a los atributos de la misma



Subclases : **Circle / Triangle**

Circle.java

```
public class Circle extends Figure {
    private double radius;

    public Circle(double x, double y, double r) {
        super(x, y);
        radius = r;
    }

    public String toString() {
        return "Circle:\n\t" +
            super.toString() +
            "\n\tRadius: " + radius;
    }
}
```

La palabra reservada **extends** se utiliza para indicar que **Circle** hereda de la clase **Figure**

La palabra reservada **super** se utiliza para referirse a elementos de la clase padre. En este caso al constructor de la superclase.

También se puede utilizar **super** para invocar un método de la clase padre sobrecargado por la subclase. En este caso al método **toString** ya implementado en la clase padre (imprime las coordenadas de posición de la figura)

Crear un nuevo projecto **BlueJ**

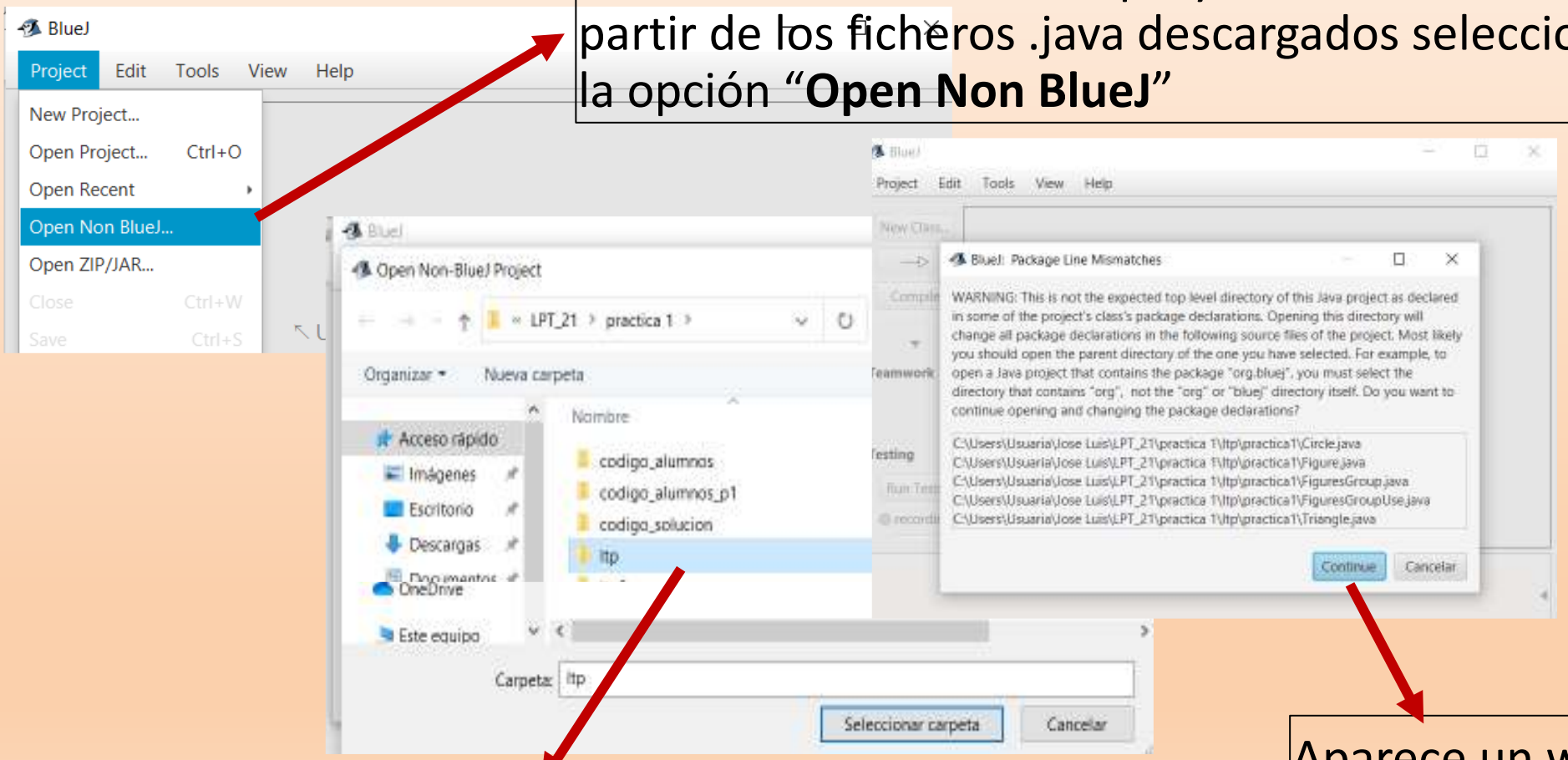
Ejercicio 1: Crea un proyecto **BlueJ** de nombre **ltp**. En el mismo, crea un paquete de nombre **practica1**. Añade a este paquete las clases que te proporcionamos en **Poliformat** como punto de partida del trabajo de esta práctica (fichero **"practica1.rar"**).

El documento proporcionado en **Poliformat, "practica1.rar"** contiene los ficheros **.java** con una primera aproximación a la solución del problema.
Entre otros ficheros podemos encontrar: **Figure.java**, **Circle.java** y **Triangle.java,** que contienen la definición de la clase **Figure** y las subclases **Circle** y **Triangle** respectivamente.

Directamente a partir de los ficheros.java (I)

Si suponemos que estos ficheros se encuentran en una carpeta **“…/ltp/practica1”…**

Podemos crear un nuevo proyecto directamente a partir de los ficheros .java descargados seleccionando la opción “**Open Non BlueJ**”

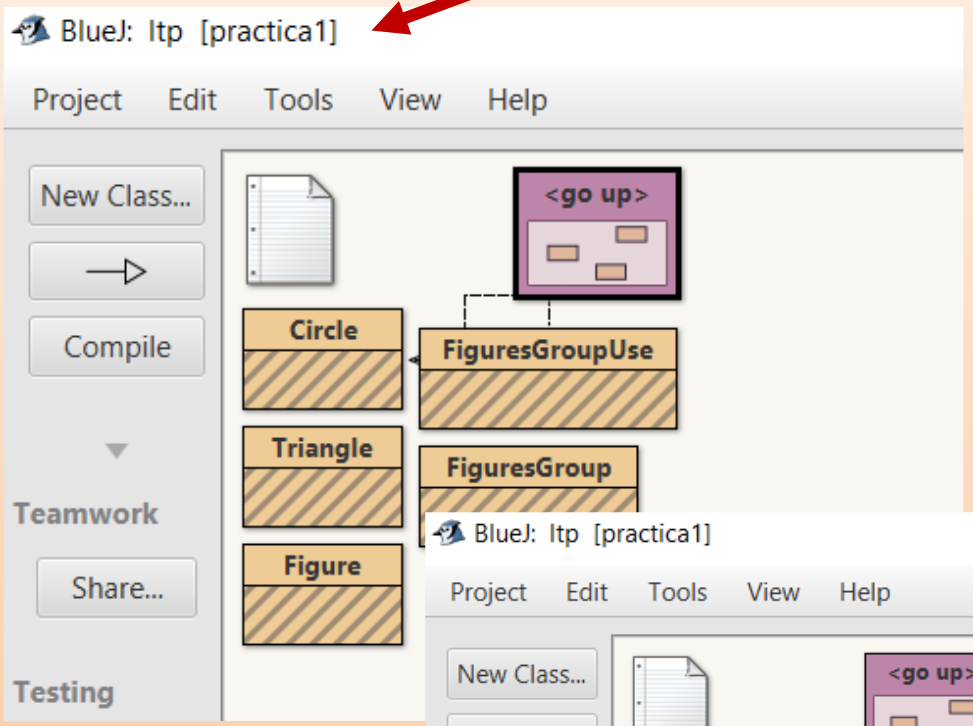


Seleccionamos la carpeta “**ltp**”

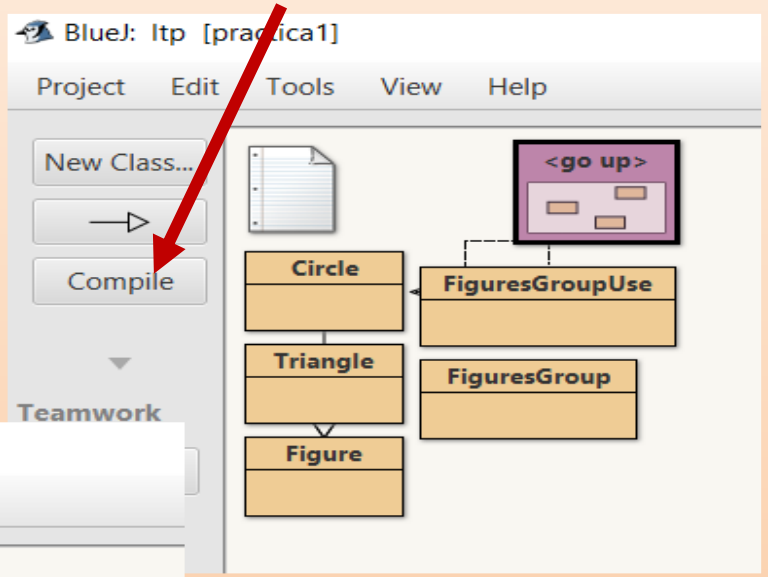
Aparece un warning, le damos a **continuar**

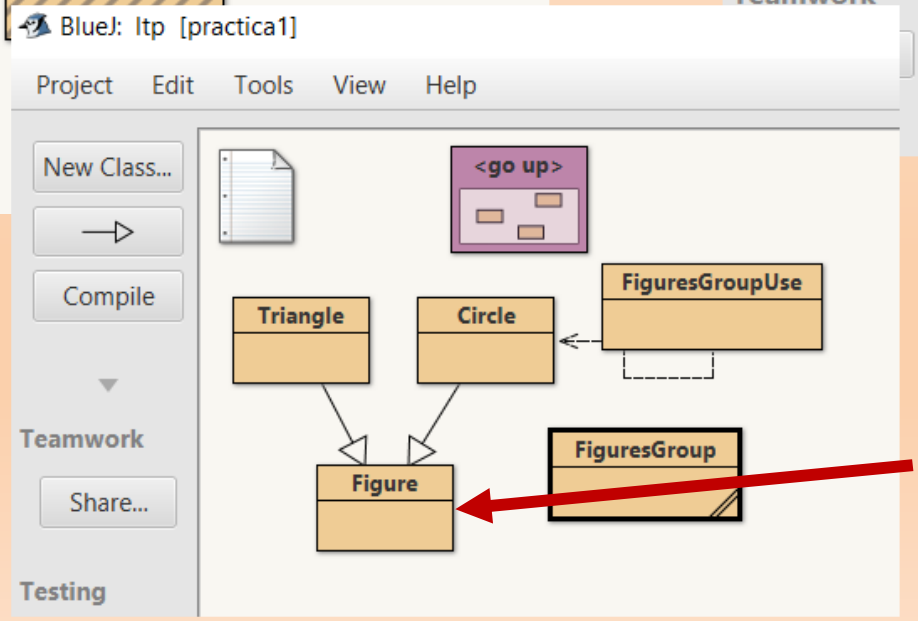
Directamente a partir de los ficheros.java (II)

Si todo ha ido bien deberá aparecer el contenido del paquete **practica1** con los 5 ficheros descargados



Si le damos la opción de **"Compile"** las clases se representarán sin sombreado

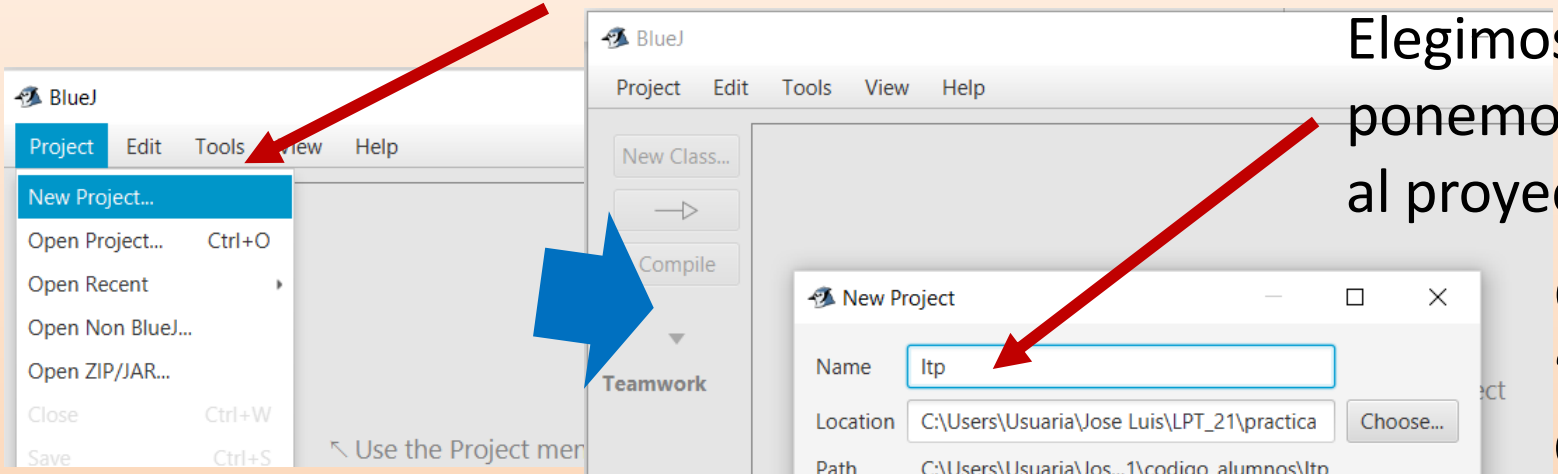




Moviendo las diferentes clases podemos ver las relaciones de dependencia entre **Figure** y sus subclases

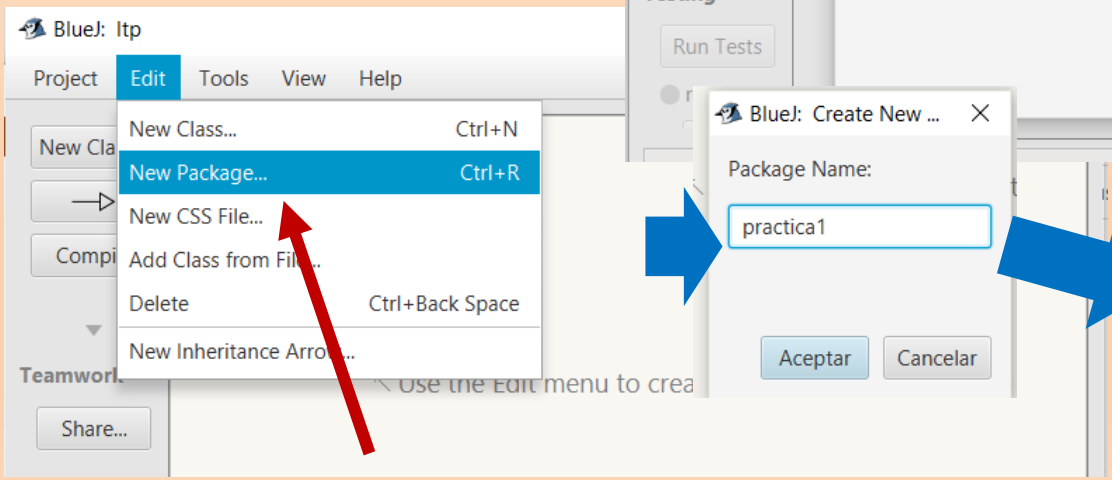
Creación manual de un nuevo proyecto

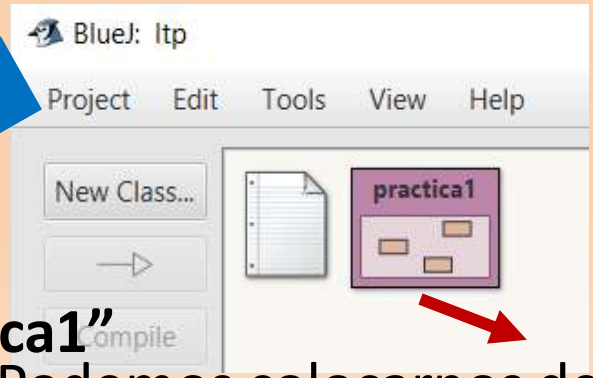
Si la carpeta **“…/ltp/practica1”** no está creada, podemos hacerlo desde el propio **BlueJ** con la opción **“New Project”**

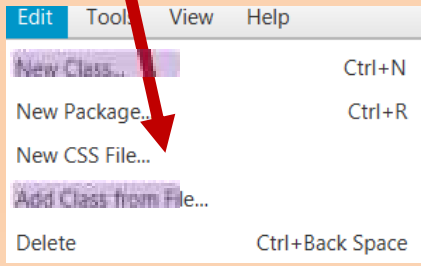


Elegimos la ubicación y ponemos el nombre **“ltp”** al proyecto

Con las opciones **“New Class”** o **“Add Class from File”** se pueden añadir las clases una a una.

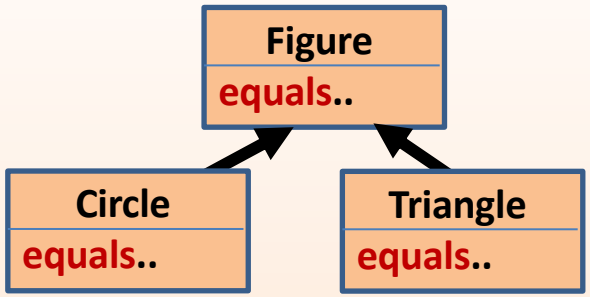






Creamos un nuevo paquete con nombre **“practica1”**

Podemos colocarnos dentro del mismo



Sobreescribir el método **equals**

Ejercicio 2: El método **equals(Object)** definido en la clase **Figure** establece que dos figuras son iguales cuando tienen la misma posición. Debemos refinar este método para las subclases **Circle** y **Triangle**: Sobrescribe el método **equals(Object)** para las clases **Circle** y **Triangle** de manera que reutilices lo ya implementado en la clase **Figure**.

equals en la clase **Figure**

```
public boolean equals(Object o) {
    if (!(o instanceof Figure)) { return false; }
    Figure f = (Figure) o;
    return x == f.x && y == f.y;
}
```

Ejercicio 2: completar **equals** en la clase **Circle** y **Triangle** (**PISTA:** hay que utilizar **super.equals**)

```
public class Circle extends Figure {
    private double radius;

    public boolean equals(Object o) {
```

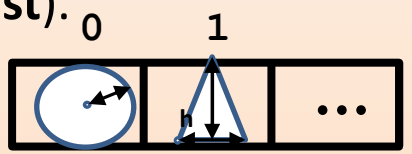
Ejercicio 2: Completar

Grupos de figuras: Clase **figuresGroup**

FiguresGroup.java

```
public class FiguresGroup {
    private static final int NUM_FIGURES = 10;
    private Figure[] figuresList = new Figure[NUM_FIGURES];
    private int numF = 0;

    public void add(Figure f) { figuresList[numF++] = f; }

    public String toString() {
        String s = "";
        for (int i = 0; i < numF; i++) {
            s += "\n" + figuresList[i];
        }
        return s;
    }

    private boolean found(Figure f) {
        for (int i = 0; i < numF; i++) {
            if (figuresList[i].equals(f)) return true;
        }
        return false;
    }
}
```

Se utiliza un array de elementos de la clase **Figure** como estructura de datos para almacenar el grupo de figuras (**figuresList**).



El atributo **numF** contiene el número de figuras (al inicio 0)

El método **add** añade un objeto de clase **Figure**

El método **toString** sobreescribe este método de la clase **Object**

Se invoca el método **toString** específico: el de **Figure,** el de **circle** o el de **Triangle**

El método **found** comprueba si una figura está contenida en el grupo (en el array **figuresList**)

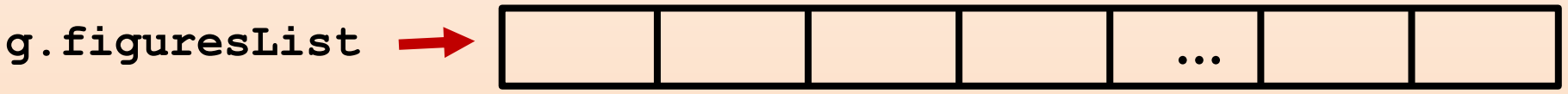
Se invoca el método **equals** específico dependiendo del tipo de figura en **figuresList[i]**

Polimorfismo de enlace dinámico: método **found** (I)

Veamos un ejemplo de uso del método **found** de la clase **FiguresGroup**, mediante una secuencia de instrucciones **Java:**

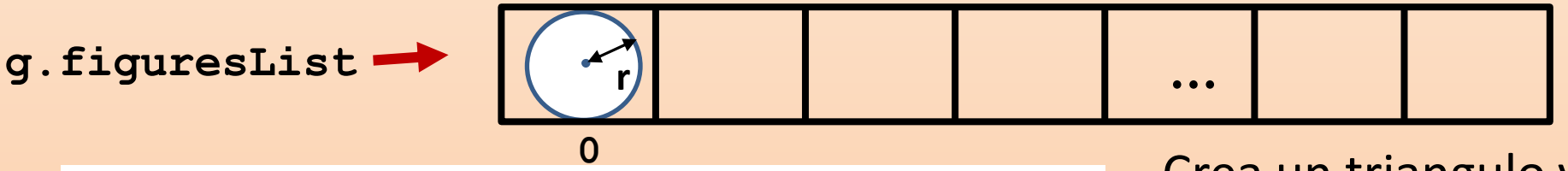
```
FiguresGroup g = new FiguresGroup();
```

Crea un grupo de figuras vacío **g**



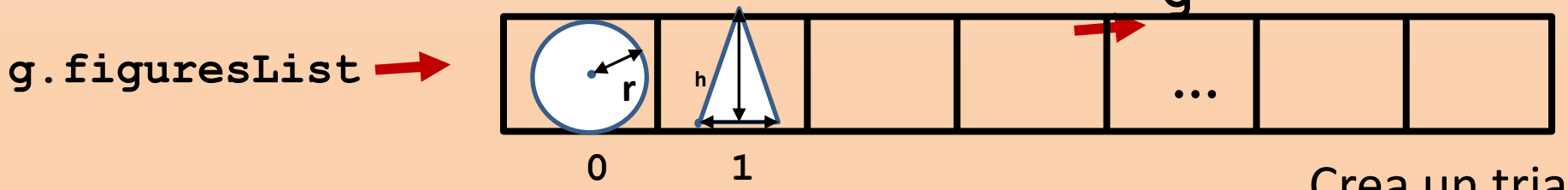

```
g.add(new Circle(10, 5, 3.5));
```

Crea un círculo y lo añade a **g**



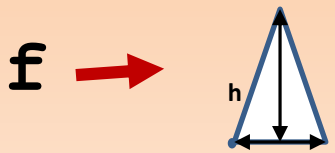

```
g.add(new Triangle(10, 5, 6.5, 32));
```

Crea un triangulo y lo añade a **g**



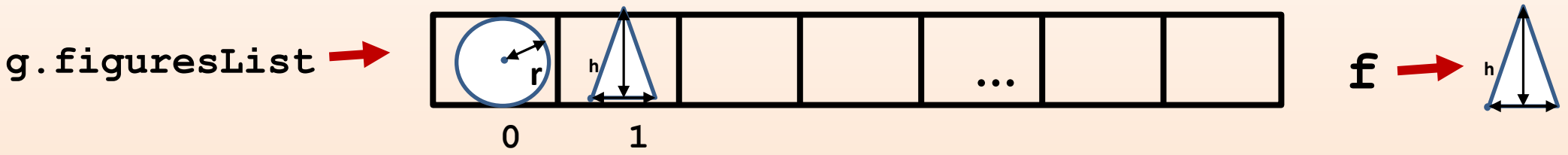

```
Triangle f = new Triangle(10, 5, 6.5, 32);
```

Crea un triangulo lo almacena en la variable **f**



Polimorfismo de enlace dinámico: método **found** (II)

Dados los valores del grupo de figuras **g** y del triángulo **f**



podemos invocar el método **found(Figure):**

```
System.out.println (g.found(f));
```

Comprueba si la figura **f** está contenida en el grupo **g** y muestra el resultado

Si hacemos una traza del método **found** se producirán dos llamadas al método **equals:**

```
private boolean found(Figure f) {
    for (int i = 0; i < numF; i++) {
        if (figuresList[i].equals(f)) return true;
    }
    return false;
}
```

i=0

i=1

```
g.figureList[0].equals(f)
```

```
g.figureList[1].equals(f)
```

Circle
...
equals

Invoca el método **equals** de **circle.** El resultado es **False**

Triangle
...
equals

Invoca el método **equals** de **Triangle.** El resultado es **True**

Creando un método **main:** clase **FiguresGroupUse**

FiguresGroupUse.java

```
public class FiguresGroupUse {
    public static void main(String[] args) {
        FiguresGroup g = new FiguresGroup();
        g.add(new Circle(10, 5, 3.5));
        g.add(new Triangle(10, 5, 6.5, 32));
        System.out.println(g);
    }
}
```

Crea un grupo de figuras vacío (variable **g**)

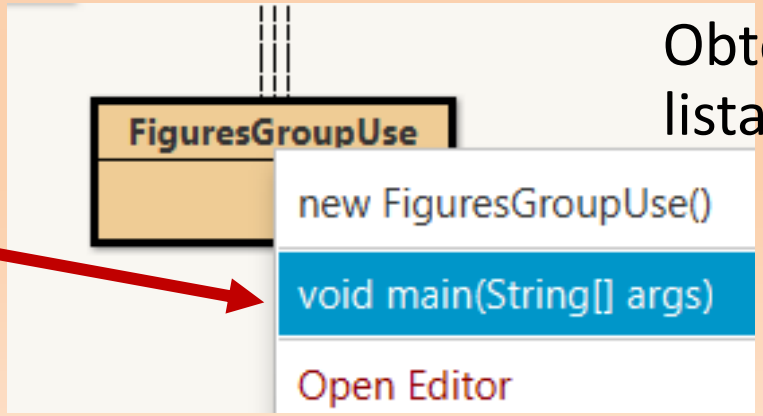
Se añade un **Circle** a **g**

Se añade un **Triangle**

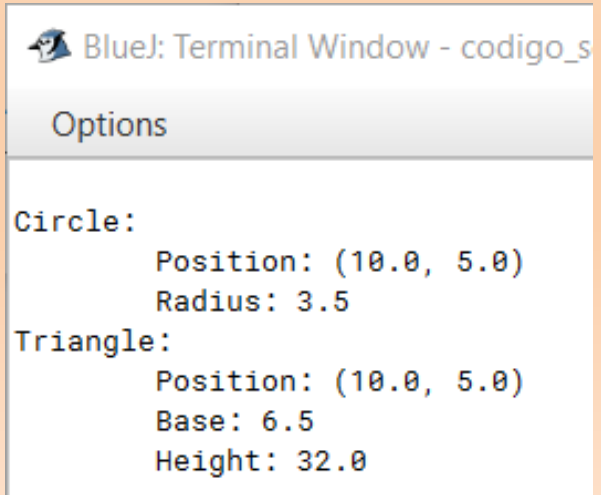
Se ejecuta **g.toString()**

Tal y como está implementada la clase **FiguresGroup,** se podría añadir a **FiguresGroupUse** la línea: "`g.add(new Figure(10, 4));`"
Es decir, se puede añadir una figura sin ninguna geometría al grupo.

Al ejecutar el **main**



Obtenemos el listado del grupo



Ampliando la clase **figuresGroup:** métodos **included(FiguresGroup)** y **equals(Object)**

```
public class FiguresGroup {
    .......
    private boolean included(FiguresGroup g) {
        for (int i = 0; i < g.numF; i++) {
            if (!found(g.figuresList[i])) return false;
        }
        return true;
    }

    public boolean equals(Object o) {
```

Ejercicio 3: Completar

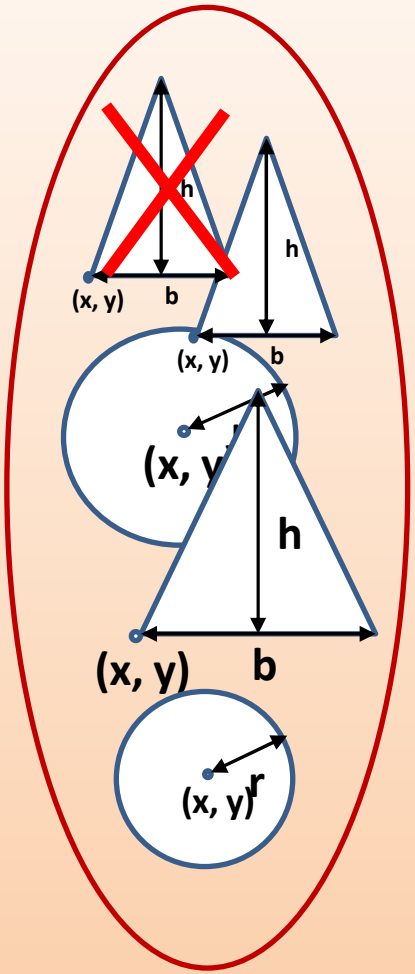
El método **included** es privado, pero puede ser utilizado dentro de la clase. Comprueba si el grupo **g** está contenido en **this**

Ejercicio 3: Sobrescribe el método **equals(Object)** para la clase **FiguresGroup**, teniendo en cuenta que dispones de los métodos **included(FiguresGroup)** y **found(Figure)** en la misma clase. Prueba el método implementado invocándolo en **FiguresGroupUse** y comparando objetos entre sí.

PISTA: dos objetos **`FiguresGroup fg1`** y **`fg2`** son iguales cuando **TODOS** los elementos de **`fg1`** están en **`fg2`** y **viceversa.**

Creando grupos **FiguresGroup** conjuntos

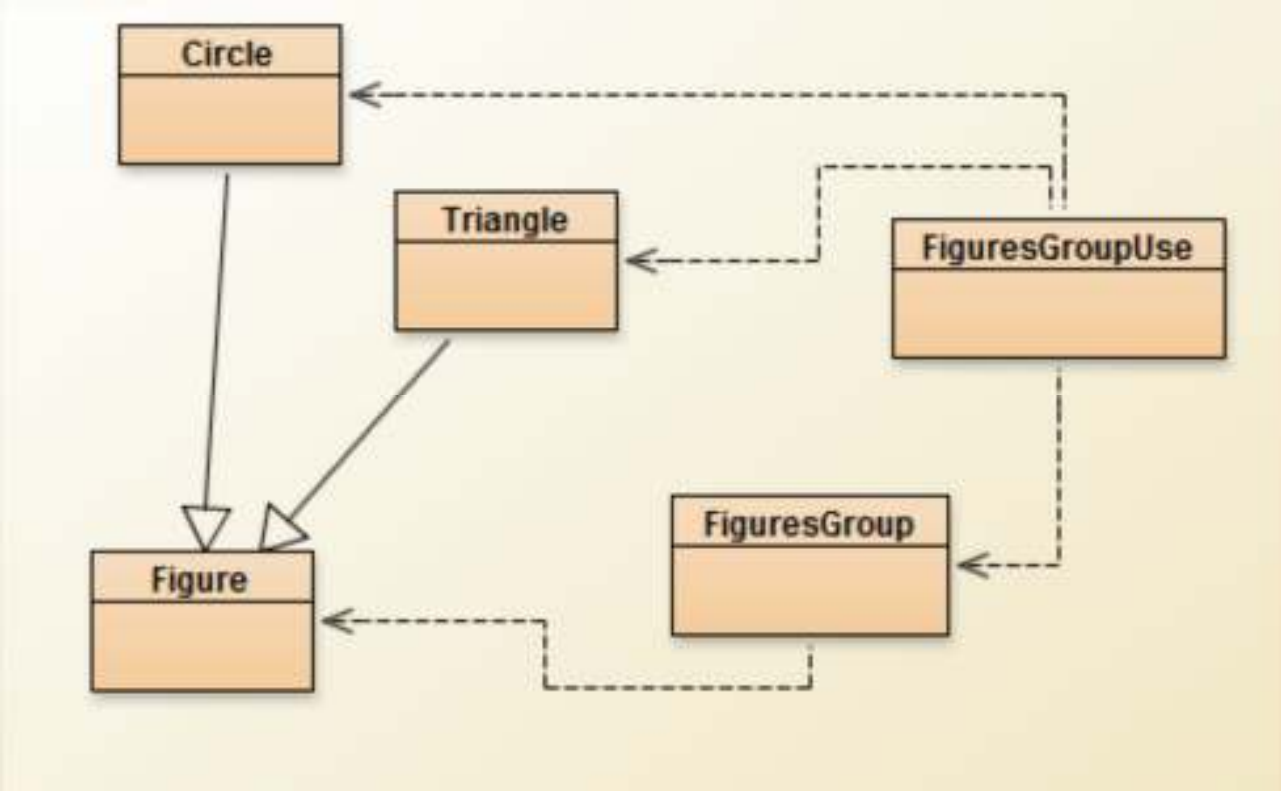
```
public class FiguresGroup {
    private static final int NUM_FIGURES = 10;
    private Figure[] figuresList = new Figure[NUM_FIGURES];
    private int numF = 0;

    public void add(Figure f) { figuresList[numF++] = f; }

    private boolean found(Figure f) {
        for (int i = 0; i < numF; i++) {
            if (figuresList[i].equals(f)) return true;
        }
        return false;
    }
```



Ejercicio 4: Considera los cambios que realizarías en los métodos **add** y **equals** de la clase **FiguresGroup** si los grupos de figuras fueran conjuntos, es decir, sin elementos repetidos. No es necesario que lo implementes.

PISTA: bastaría con modificar **add** comprobando, antes de añadir el elemento al vector **figuresList,** que esté no se encuentra ya en el grupo.

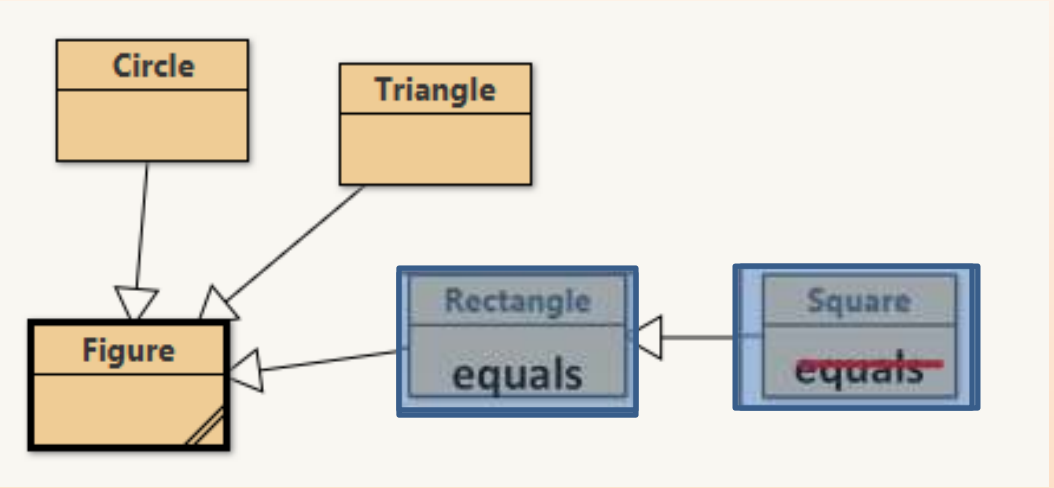
Relaciones entre las diferentes clases



Recordamos que las líneas continuas representan relaciones de herencia, mientras que las líneas discontinuas indican uso.

Es destable el hecho de que **FiguresGroup** no usa a las clases relacionadas con figuras geométricas. Solo usa la clase **Figure**, lo que permitirá añadir nuevas figuras geométricas sin necesidad de modificar su código.

Añadiendo figuras: clases **Rectangle** y **Square**



Ejercicio 5: Para representar la figura geométrica Rectángulo, define una clase **Rectangle** con atributos **base** y **height** de tipo **double**, que derive de **Figure**, y con los mismos métodos que **Circle** y **Triangle**.
¿Cambia en algo **FiguresGroup**?
Añade un rectangulo al grupo de figuras definido en la clase **FiguresGroupUse** para comprobarlo.

Ejercicio 6: Para representar la figura geométrica **Cuadrado**, define una clase **Square**, sin atributos, que derive de **Rectangle**, y con la misma funcionalidad que su superclase.
¿Has necesitado sobrescribir algún método de **Rectangle**? ¿Por qué?

3. Solución 2 : usando clases abstractas

La solución planteada hasta ahora permite que objeto **FiguresGroup** contenga objetos **Figure** (sin forma geométrica asociada).

Esto se podría evitar modificando la clase **FiguresGroup** de forma que, antes de añadir un elemento, se compruebe si pertenece a una de las clases con forma geométrica.

Sin embargo, esta solución requiere la modificación de la clase **FiguresGroup** cada vez que añadamos nuevas figuras, dificultando el mantenimiento de nuestra aplicación. **(Ver Ejercicio 7)**.

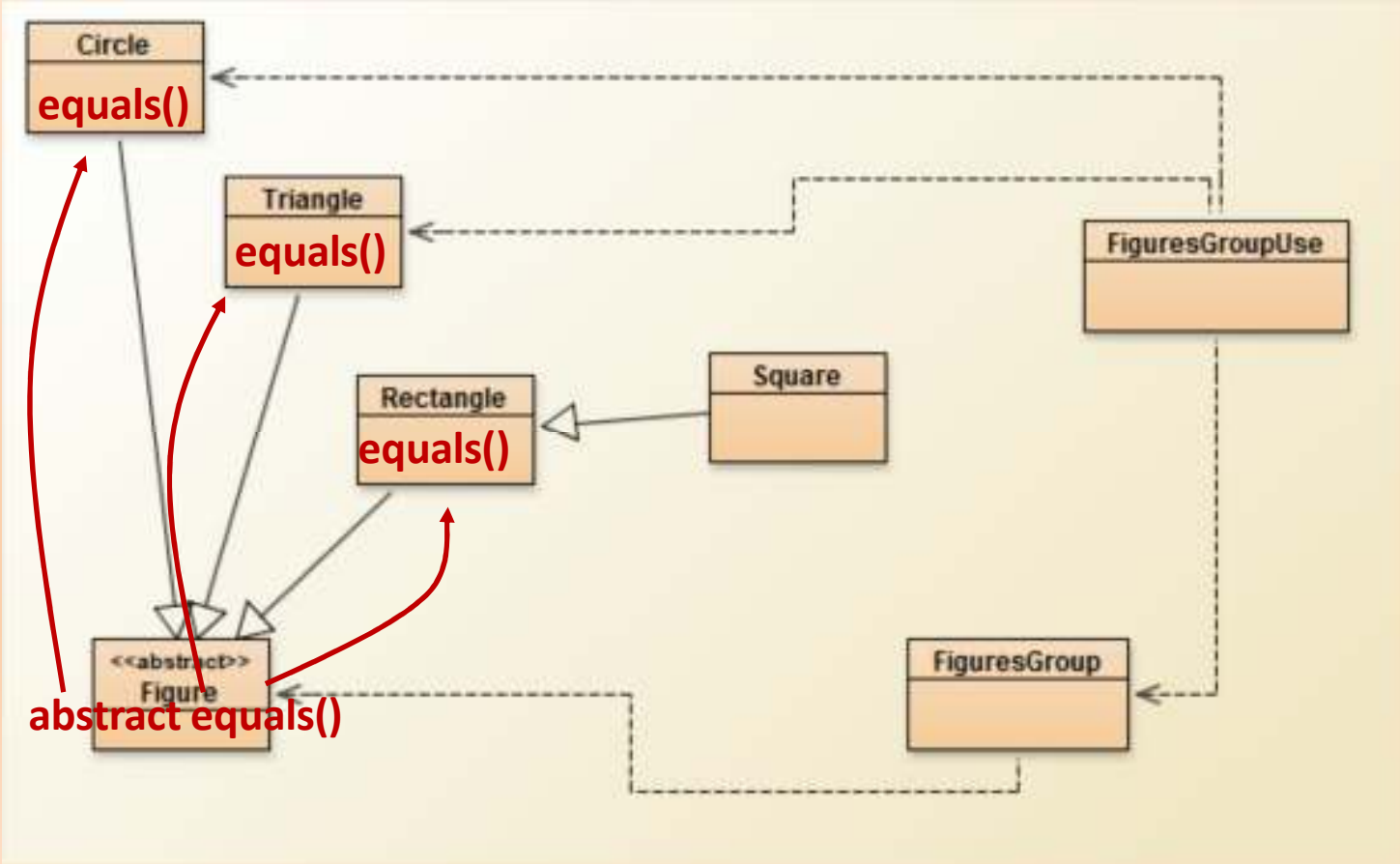
Otra opción más sencilla consiste en definir la clase **Figure** como abstracta, lo que impide que se puedan crear objetos de esta clase. Para ello simplemente hay que añadir la palabra reservada **abstract** a la cabecera de la clase **Figure** de la siguiente forma:

```
public abstract class Figure
```

3. Solución 2 : usando clases abstractas

Las clases abstractas, como lo es ahora, **Figure,** pueden definir métodos abstractos. Un método abstracto se define con su cabecera pero no se implementa.

Las clases herederas, **Circle, Triangle** y **Rectangle** se verán obligadas a implementar todos los métodos abstractos definidos en **Figure.**



Definiendo un método abstracto: **area()**

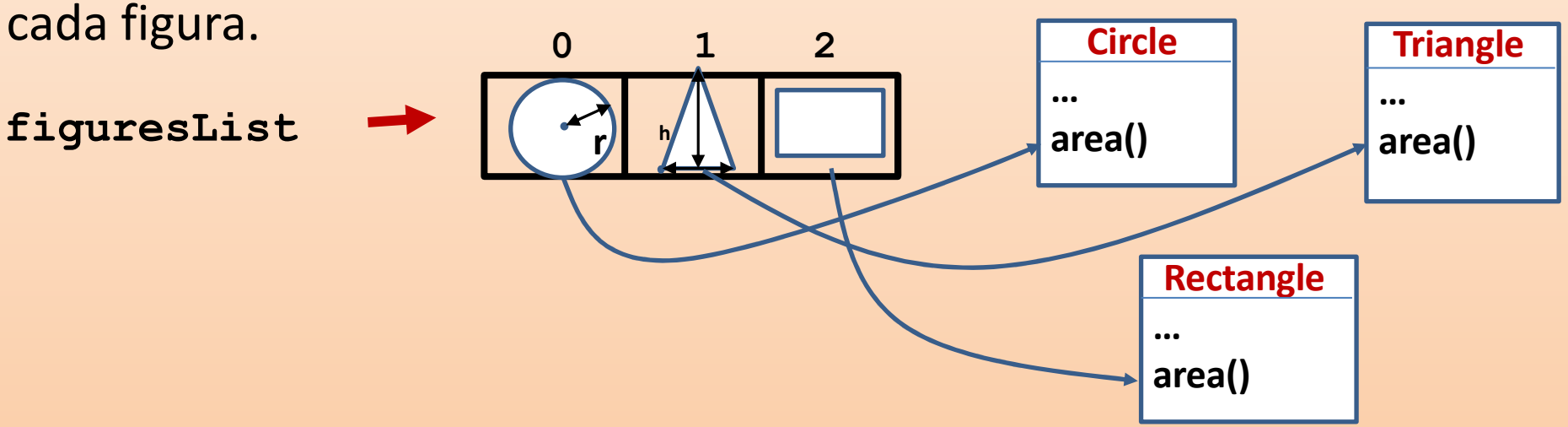
Ejercicio 8: Se desea que todas las figuras dispongan de un método para calcular su área. Define en la clase **Figure** un método abstracto **area()** que devuelva un valor de tipo **double**.
PISTA: añade a **Figure** la línea "`public abstract double área();`"
Tras esta modicacion de la clase **Figure**, las clases derivadas dejaran de compilar, con el mensaje de error**:**
"... is not abstract and does not override abstract method area() in Figure".

Ejercicio 9: Resuelve el problema implementando el método **area()** en cada una de las subclases de **Figure**. Modifica el código de **FiguresGroupUse** para calcular y visualizar el área de las figuras que se crean en él.

PISTA: Podemos calcular las áreas mediante las siguientes expresiones:
Área del círculo = `Math.PI * radius * radius`
Área del triangulo = `base * height / 2`
Área del rectangulo = `base * height`

Suma de áreas y máxima área de un grupo de figuras: métodos **area()** y **greatestFigure()**

Ejercicio 10: Define un método **area()** en la clase **FiguresGroup** que devuelva la suma de las áreas de las figuras de un grupo. Para ello recorre todas las figuras referenciadas en las componentes del atributo **figuresList** desde la posición cero hasta la posición **numF-1** aplicando el método **area()** a cada figura.



Ejercicio 11: Define un método **greatestFigure()** en la clase **FiguresGroup** que devuelva la figura del grupo cuya área sea mayor. De nuevo, tendrás que recorrer todas las figuras en el array **figuresList** y aplicar el método **area()** a cada figura.