Ministerio de Producción
Presidencia de la Nación

Ministerio de Educación y Deportes

Subsecretaría de Servicios Tecnológicos y Productivos

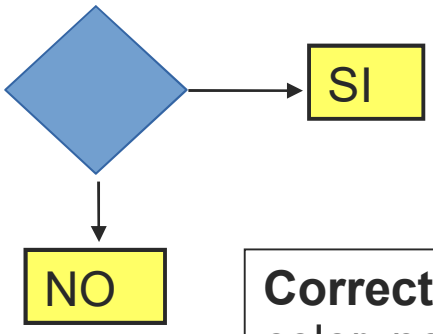
Clases abstractas

Programación

- Clases abstractas
- La palabra clave **abstract**
- Métodos abstractos
- Clases abstractas y polimorfismo
- Ejercicios prácticos en PC

Clases abstractas

- En Java, es posible definir clases que representan un **concepto abstracto** y como tal **no pueden ser instanciadas**
- Pensemos en un ejemplo del mundo real: el concepto de “comida”. Si definimos una clase llamada Comida, ¿es posible crear instancias?



¿Qué forma tiene? ¿Qué color? ¿Qué sabor? ¿Qué temperatura?

¿En qué momento del día se come? ¿Cómo se sirve?

No es posible determinarlo!!!

Correcto! La comida como **concepto es abstracto**, no tiene forma, no tiene color, no tiene sabor, no tiene temperatura... **solo representa un conjunto de sustancias alimenticias que se consumen en diferentes momentos del día pero de la que no podemos determinar otras características**

Clases abstractas

La comida es un **concepto abstracto** que representa las sustancias alimenticias que consumimos en diferentes momentos del día

La clase **Comida** es una clase abstracta

Clases abstractas versus concretas

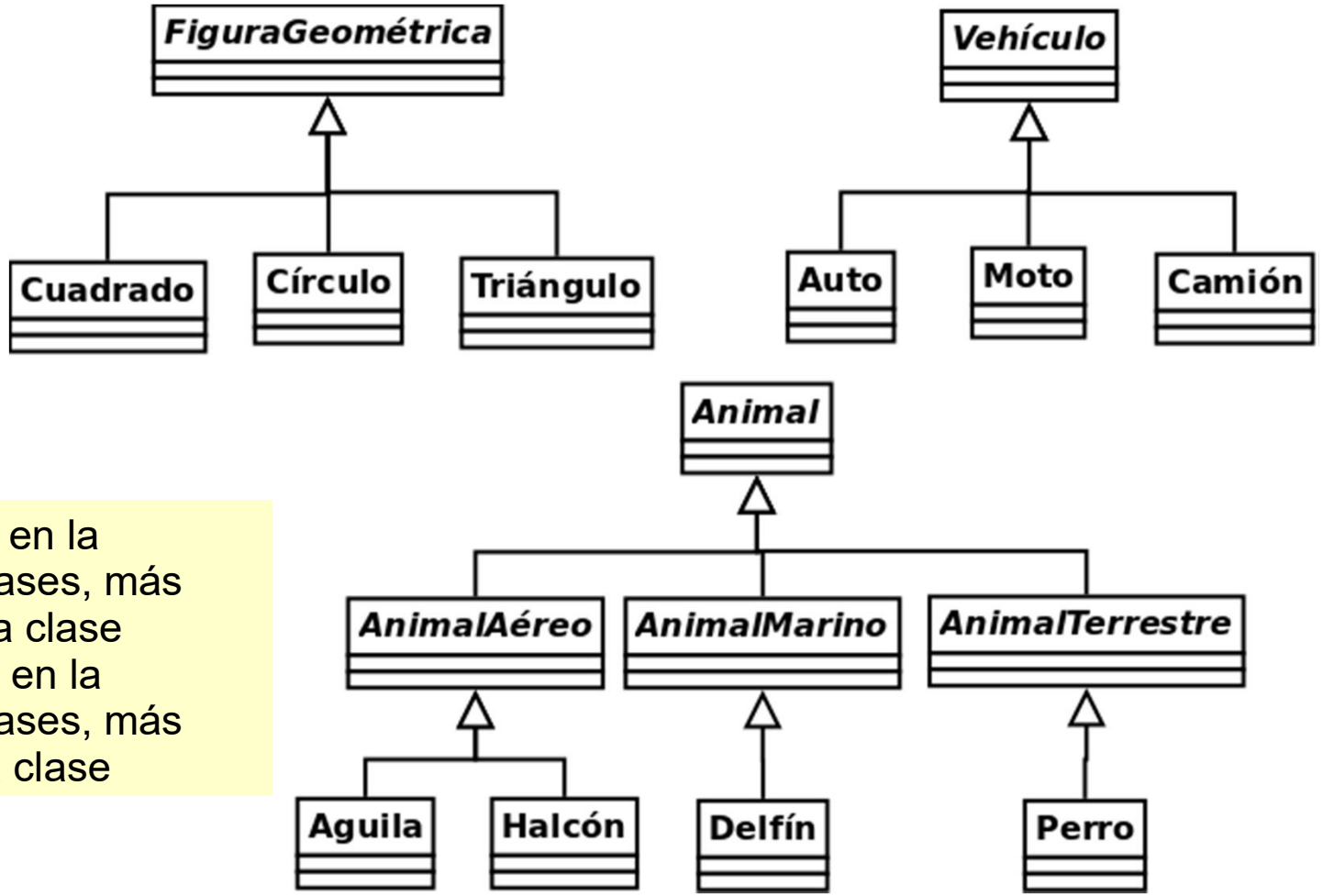
Sin embargo, si pensamos en determinadas comidas: **las frutillas**, **el pollo**, **las hamburguesas**, etc. entonces **SI** pensamos en conceptos concretos pues se trata de tipos de comidas que tienen características propias como sabor, color, temperatura, forma...

En este caso, podemos pensar en objetos concretos con características propias:

- Cada pollo tiene su propio peso, sabor y temperatura de acuerdo a la preparación
- Cada hamburguesa tiene su propio sabor de acuerdo a si es casera o no y al tipo de carne con la que se prepare, pueden ser de diferente tamaño y forma, etc.

Son conceptos concretos

Clases abstractas: Otros ejemplos



- A mayor altura en la jerarquía de clases, más **abstracta** es la clase
- A menor altura en la jerarquía de clases, más **concreta** es la clase

Clases abstractas en Java

- Una clase abstracta es una clase que solamente puede ser **extendida**, pero **no puede ser instanciada** (no puede usarse junto al operador **new**)
- Para declarar una **clase abstracta** se antepone la palabra clave **abstract** a la palabra clave **class**.

```
public abstract class Comida {
  // cuerpo de la clase
}
```

```
public abstract class FiguraGeométrica {
  // cuerpo de la clase
}
```

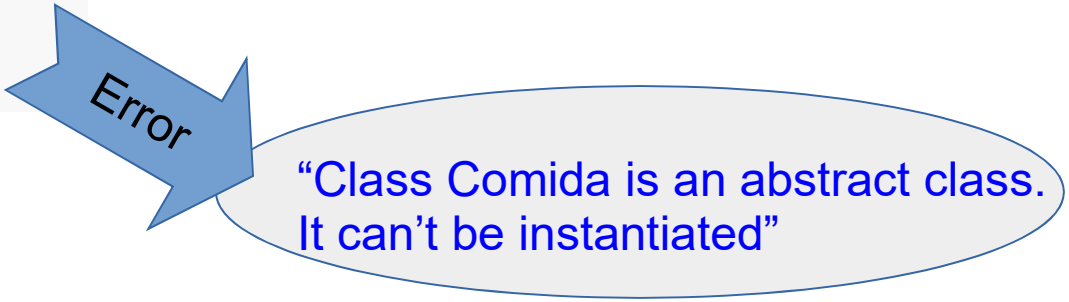
```
public abstract class Vehículo {
  // cuerpo de la clase
}
```

```
public abstract class Animal {
  // cuerpo de la clase
}
```

Clases abstractas en Java

- Si se intenta crear objetos de una **clase abstracta**, fallará la compilación

```
public class TestComida {
        public static void main(String args[]) {
                new Comida();
        }
}
```



- El compilador de Java garantiza la pureza de las clases abstractas NO permitiendo crear instancias
- *Puede reproducir este error adicionando la palabra clave **abstract** a cualquier clase ya definida en un ejercicio anterior hecho en máquina*

Clases abstractas: Resumen

- En Programación Orientada a Objetos los **conceptos abstractos** se modelan mediante una **clase abstracta** cuya finalidad **NO** es crear instancias como en las clases que venimos implementando hasta ahora

El objetivo de una **clase abstracta** es ser **extendida** por clases concretas y definir una **interface de comportamiento común de los objetos de sus subclases**

La API de Java hace uso intensivo de clases abstractas!

Por ejemplo la clase **Number** del paquete **java.lang** representa el **concepto abstracto de los números**: de todos los números específicos (enteros, decimales) podemos obtener su valor primitivo, sin embargo de un "número" del que sabemos solamente que es un "número" **NO**. En nuestros programas creamos objetos Integer, Double, Float a los que podemos asignarles valores. **NO** creamos objetos Number.

La clase **Number** es la **superclase abstracta** de otras clases que permiten representar números concretos: **Integer, Float, Double,** etc.

Métodos abstractos y concretos

Una **clase abstracta** puede contener **métodos abstractos** y **métodos concretos**.

¿Qué es un método abstracto?

- Es un método que NO tiene implementación o código
- Se define el nombre, el tipo de retorno, la lista de argumentos, termina con ";" y se antepone la palabra clave abstract. NO tiene código

```
public abstract class FiguraGeométrica {
  public abstract int calcularArea();
  // otros métodos abstractos o concretos
}
```

¿Y para qué sirve un método sin código entonces?

→ Es la forma de definir explícitamente el comportamiento común de todos los objetos de las subclases concretas de la clase abstracta

Métodos abstractos

- ¿Pueden existir fuera de una clase abstracta? **NO**
- Si una clase incluye un método abstracto, ¿forzosamente la clase será una clase abstracta? **SI**
- ¿Es necesario declarar como **abstract** una clase que contiene algún método abstracto? **SI**

```
public class FiguraGeométrica {
  public abstract int calcularArea();
  // otros métodos abstractos o concretos
}
```

```
public abstract class FiguraGeométrica {
  public abstract int calcularArea();
  // otros métodos abstractos o concretos
}
```

Métodos abstractos

- ¿Una clase se puede declarar como abstracta y no contener métodos abstractos? **SI**

```
public abstract class FiguraGeométrica {
  public float retornarValorPi() {
    return 3.14159f;
  }
  // otros métodos concretos
}
```

- ¿Una clase que hereda de una clase abstracta puede ser no abstracta? ¿Puede ser concreta? **SI a ambas!**

  → *Ilustraremos esto un poco más adelante...*

Métodos abstractos en práctica

- Consideremos ahora una aplicación Java que dibuja por pantalla figuras geométricas. Podríamos dibujar por ejemplo **círculos**, **rectángulos**, **triángulos**, **líneas rectas**, etc. Todas las figuras geométricas pueden **dibujarse** en la pantalla, **redimensionarse**, y **moverse**, pero cada una lo hace de una manera particular (en especial dibujarse y redimensionarse)...
- Por otro lado, una figura geométrica, es un concepto abstracto, no es posible dibujarla o redimensionarla: solo sabemos que todas las figuras geométricas concretas tienen esas capacidades
- Pensemos entonces:

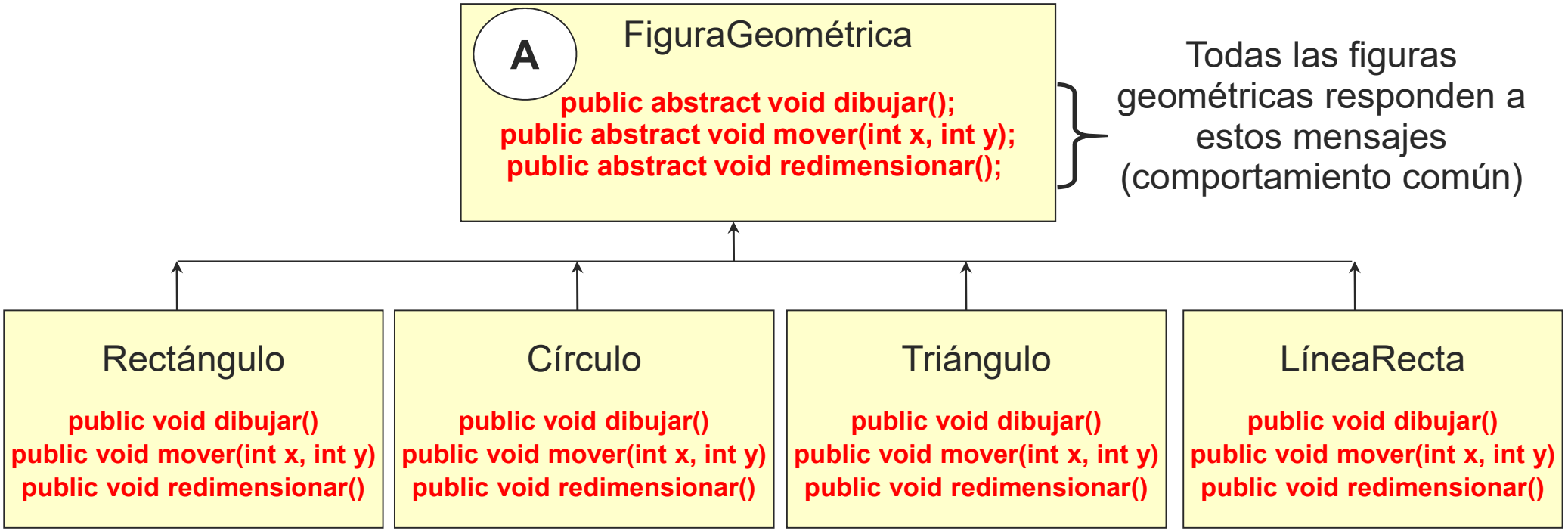
  ¿cuáles serían las clases abstractas de la aplicación?

  ¿cuáles serían las clases concretas?

  ¿cuáles serían los métodos abstractos?

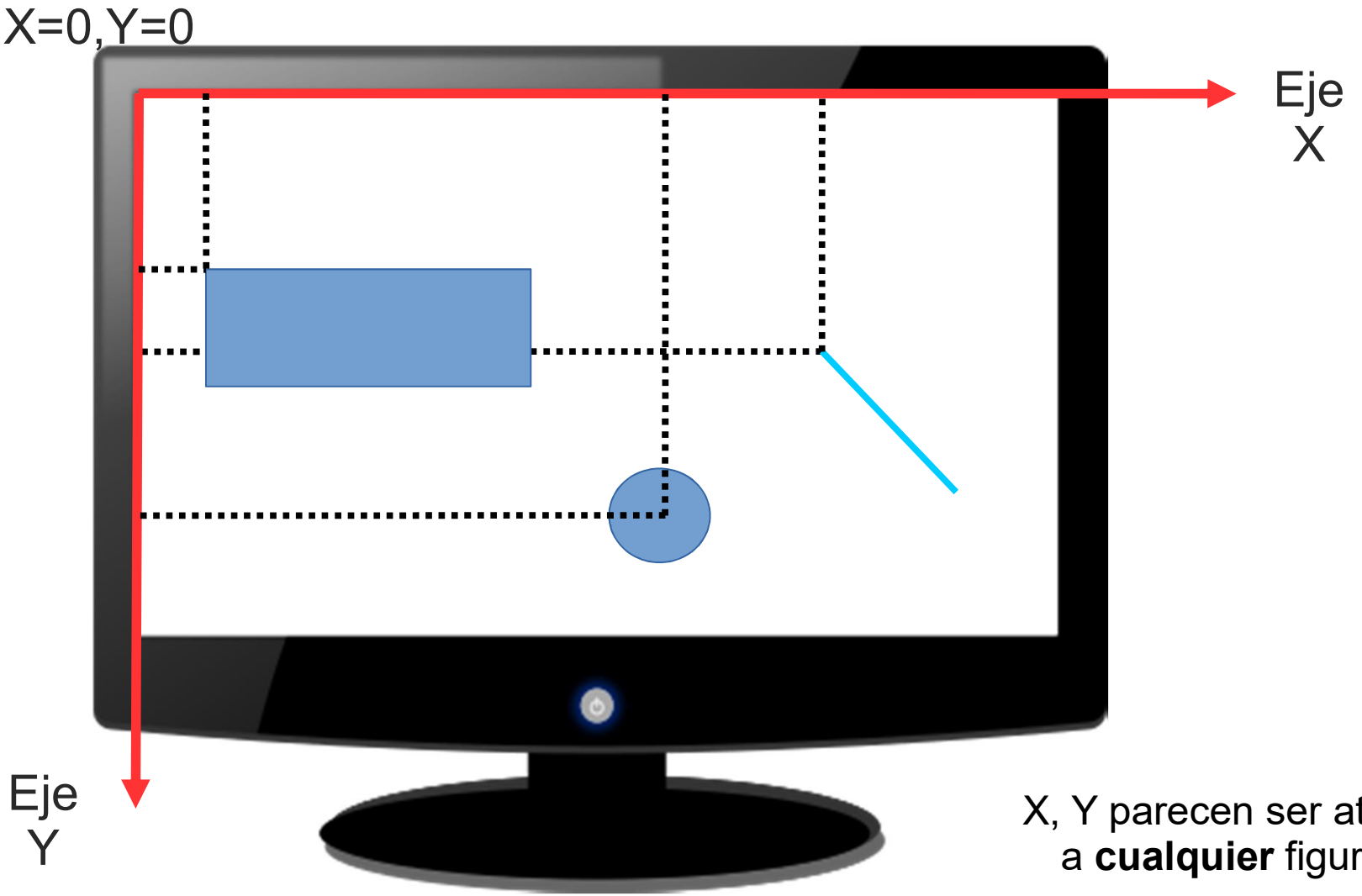
  ¿cuál sería el procedimiento de dibujado en cada caso?

Métodos abstractos en práctica



- **FiguraGeométrica** es una clase abstracta y **dibujar() mover()** y **redimensionar()** son métodos abstractos
- **Círculo**, **Rectángulo**, **Triángulo** y **LíneaRecta** son subclases concretas de **FiguraGeometrica** y proveerán una implementación concreta a cada uno de los métodos abstractos de **FiguraGeométrica**

Métodos abstractos en práctica



X, Y parecen ser atributos comunes a **cualquier** figura...

¿Las clases abstractas pueden declarar variables de instancia?

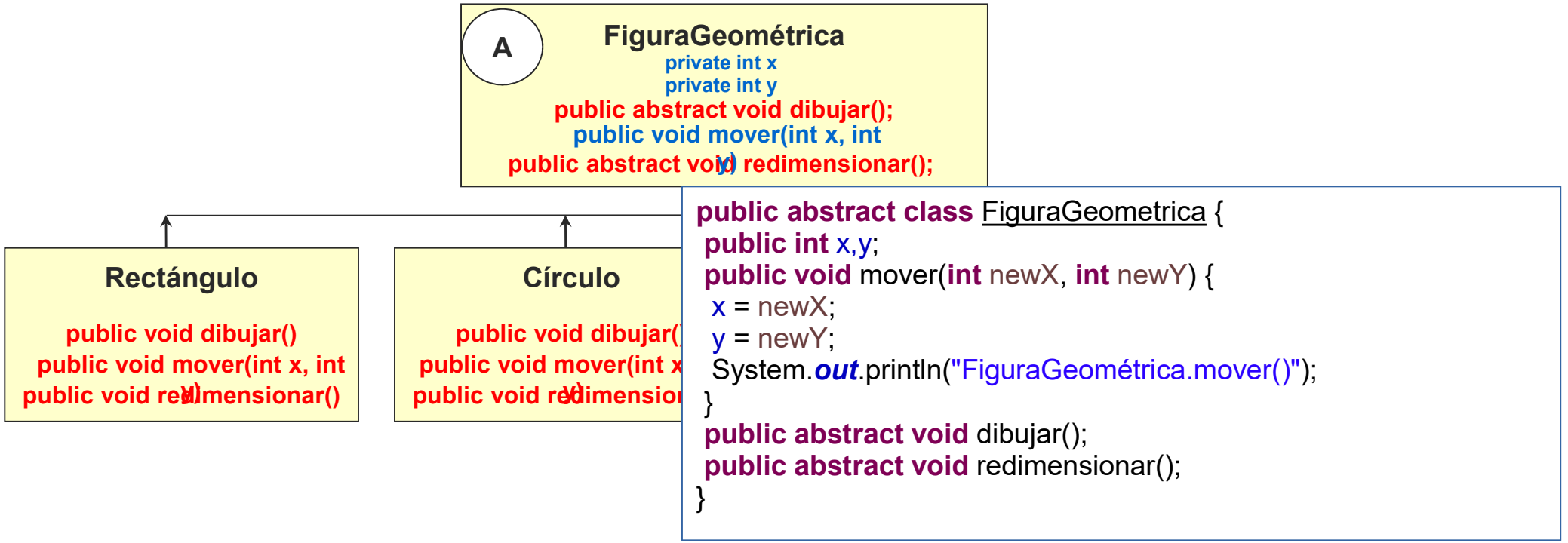
Métodos abstractos en práctica

- **Efectivamente! Las clases abstractas pueden declarar variables de instancia**

En ejemplo debajo, la clase abstracta **FiguraGeométrica** ahora declara variables de instancia que representan la **posición actual de la figura**

- ¿Las clases abstractas pueden declarar métodos concretos? **SI**

Podríamos codificar al método **mover()** de **FiguraGeométrica** y proveerle una implementación que compartirían todas las subclases concretas



Métodos abstractos y sobreescritura

- Los métodos abstractos **deben** ser sobreescritos en las subclases
- Cada subclase **concreta** de **FiguraGeometrica**, como **Rectangulo** y **Triangulo** por ejemplo **heredan** el método **mover()** y su implementación y **deben proveer una implementación** para el método **dibujar()** y **redimensionar()**:

```
abstract class FiguraGeometrica {
 public int x,y;

 public void mover(int newX, int newY) {
  x = newX;
  y = newY;
  System.out.println(
          "FiguraGeométrica.mover()");
 }

 public abstract void dibujar();
 public abstract void redimensionar();

}
```

```
class Rectangulo extends FiguraGeometrica {
 public void dibujar() {
  System.out.println("Rectangulo.dibujar()");
 }
 public void redimensionar() {
  System.out.println("Rectangulo.redimensionar()");
 }
}

class Triangulo extends FiguraGeometrica {
 public void dibujar() {
  System.out.println("Triangulo.dibujar()");
 }
 public void redimensionar() {
  System.out.println("Triangulo.redimensionar()");
 }
}
```

Métodos abstractos y sobreescritura

```
public class TestFigurasGeometricas {

 public static void main(String[] args) {
  Triangulo t = new Triangulo();
  Rectangulo r = new Rectangulo();

  t.dibujar();
  t.mover(10, 10);
  t.redimensionar();

  r.dibujar();
  r.mover(15, 20);
  r.redimensionar();
 }

}
```

¿Cuál sería la salida?

Triangulo.dibujar()
FiguraGeometrica.mover()
Triangulo.redimensionar()
Rectangulo.dibujar()
FiguraGeometrica.mover()
Rectangulo.redimensionar()

Métodos abstractos y sobreescritura

- Si una subclase no implementa un método abstracto de la superclase, heredará un método no ejecutable (sin implementación), lo que la fuerza a ser una subclase abstracta
- Un ejemplo simple sería:

```
abstract class FiguraGeometrica {
 public int x,y;

 public void mover(int newX, int newY) {
  x = newX;
  y = newY;
  System.out.println(
          "FiguraGeométrica.mover()");
 }

 public abstract void dibujar();
 public abstract void redimensionar();

}
```

```
abstract class Cuadrilatero
        extends FiguraGeometrica {
 public void redimensionar() {
  System.out.println("Cuadrilatero.redimensionar()");
 }
}

class Rectangulo extends Cuadrilatero {
 public void dibujar() {
  System.out.println("Rectangulo.dibujar()");
 }
}
```

Clases abstractas y asignación

- Al igual que con las clases concretas, podemos asignar objetos de una clase C1 a una variable de tipo C2 siempre y cuando las clases sean iguales o bien C2 sea una clase descendiente de C1
- Volviendo al código de la diapositiva anterior:

```
abstract class FiguraGeometrica {
 public int x,y;

 public void mover(int newX, int newY){...}

 public abstract void dibujar();
 public abstract void redimensionar();

}
```

```
abstract class Cuadrilatero
        extends FiguraGeometrica {
 public void redimensionar() {...}
}

class Rectangulo extends Cuadrilatero {
 public void dibujar() {...}
}
```

¿Entonces, pueden encontrar la asignación que no es posible en el siguiente código?

```
class TestFigurasGeometricas {
 public static void main(String[] args) {
  FiguraGeometrica fg = null;
  Cuadrilatero c = null;
  Rectangulo r = new Rectangulo();
  c = new Rectangulo();
  fg = new Rectangulo();
  fg = c;
  c = r;
  c = fg;
 }
}
```

Clases abstractas y polimorfismo

- El polimorfismo es la propiedad de la programación orientada a objetos de que dos objetos respondan de diferente forma al mismo mensaje. Ejemplo: objetos de la clase **Triangulo** y **Rectangulo** responden diferente a invocaciones a **dibujar()**...

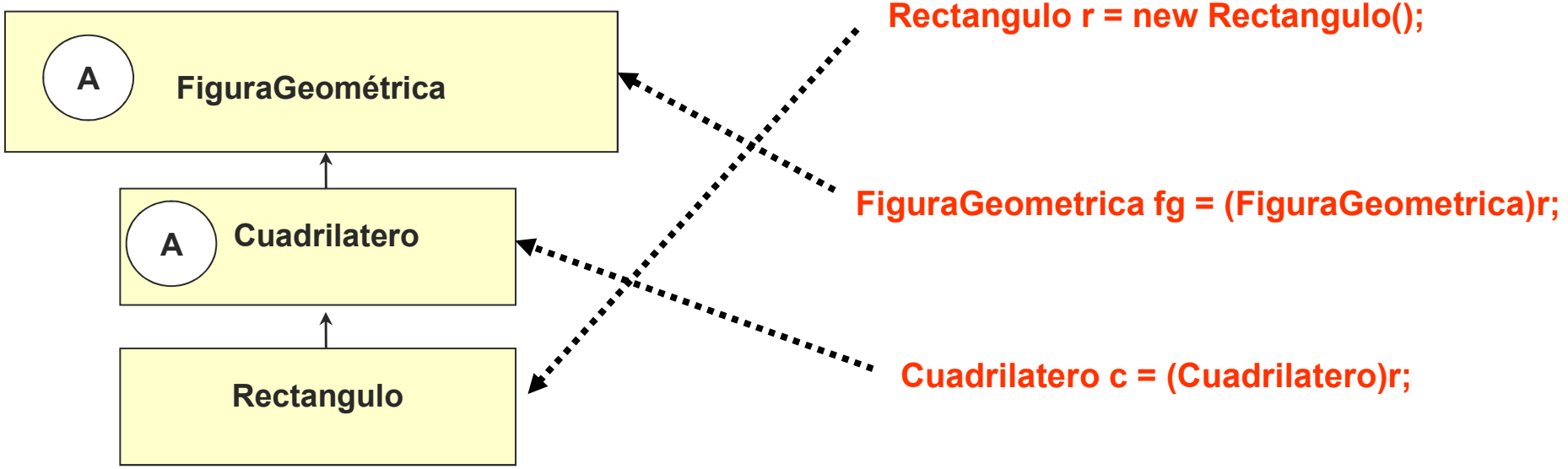
```
public class TestFiguraGeometricas2 {

 public static void main(String[] args) {
   FiguraGeometrica figuras[] = new FiguraGeometrica[5];
   FiguraGeometrica f;
   for (int i = 0; i < figuras.length; i++) {
      if (i % 2 == 0)
         f = new Circulo();
      else
        f = new Rectangulo();
      figuras[i] = f;
   }
   for (FiguraGeometrica g : figuras) {
      g.dibujar();
   }
 }
}
```

FiguraGeometrica es la **superclase común** de todas las figuras geométricas: cualquier figura puede ser asignada a la variable o elementos del arreglo, **sin necesidad de hacer casting**

Es posible invocar a **dibujar()** de cualquier objeto **FiguraGeometrica aunque la clase FiguraGeometrica no defina el código de dichos métodos**, sino que se invoca al método **dibujar()** concreto

Y si combino clases abstractas con *casting?*

- En Java es posible usar también casting en combinación con clases abstractas, “casteando” objetos de una subclase concreta a objetos de una clase abstracta…
- Veamos ejemplos:

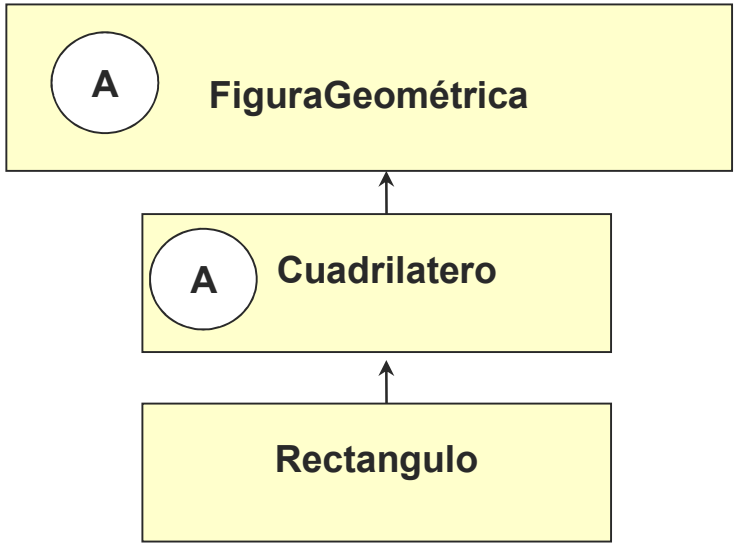


¿Funcionaría una definición como

FiguraGeometrica f = (FiguraGeometrica)c?

Y si combino clases abstractas con *instanceof?*

- El operador **instanceof** aplicado a un objeto de una clase que tiene algún ancestro abstracto también aplica!
- Veamos ejemplos:

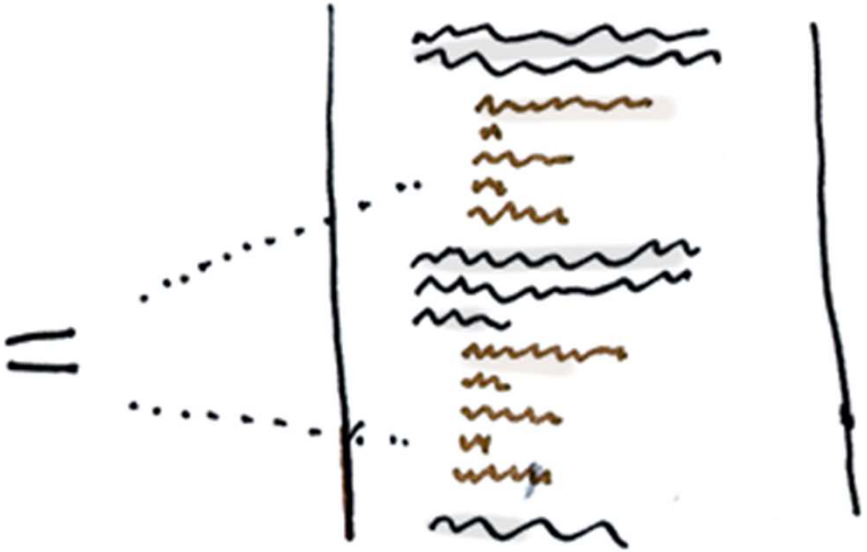



```
public class TestFigurasGeometricasInstanceOf {
 public static void main(String[] args) {
  Rectangulo r = new Rectangulo();
  Cuadrilatero c = (Cuadrilatero)r;

  System.out.println(c instanceof FiguraGeometrica);
  System.out.println(r instanceof FiguraGeometrica);
  System.out.println(r instanceof Cuadrilatero);

 }
}
```

La salida sería “true” en todos los casos...

Clases abstractas y duplicación de código

- La **duplicación de código** es la situación donde una misma porción de código fuente ocurre varias veces en un mismo programa
- Esta situación es indeseable por varias razones, incluyendo:
  - Un error en una de las porciones se manifiesta en las otras
  - Puede corregirse el error en algunas porciones y olvidarse en otras
  - El código es más largo y difícil de leer



Clases abstractas y duplicación de código

- Ejemplo:

```
public class DuplicacionTest {
 public static void main(String[] args) {
  // goles por mes anotados por dos delanteros
  int[] arreglo_goles_jugadorA = new int[10];
  int[] arreglo_goles_jugadorB = new int[10];

  int suma_a = 0;
  for (int i = 0; i < 10; i++)
   suma_a += arreglo_goles_jugadorA[i];
  int promedio_anualA = suma_a / 10;

  int suma_b = 0;
  for (int i = 0; i < 10; i++)
   suma_b += arreglo_goles_jugadorB[i];
  int promedio_anualB = suma_b / 10;

  System.out.println(promedio_anualA);
  System.out.println(promedio_anualB);
 }
}
```

Porciones duplicadas!!!

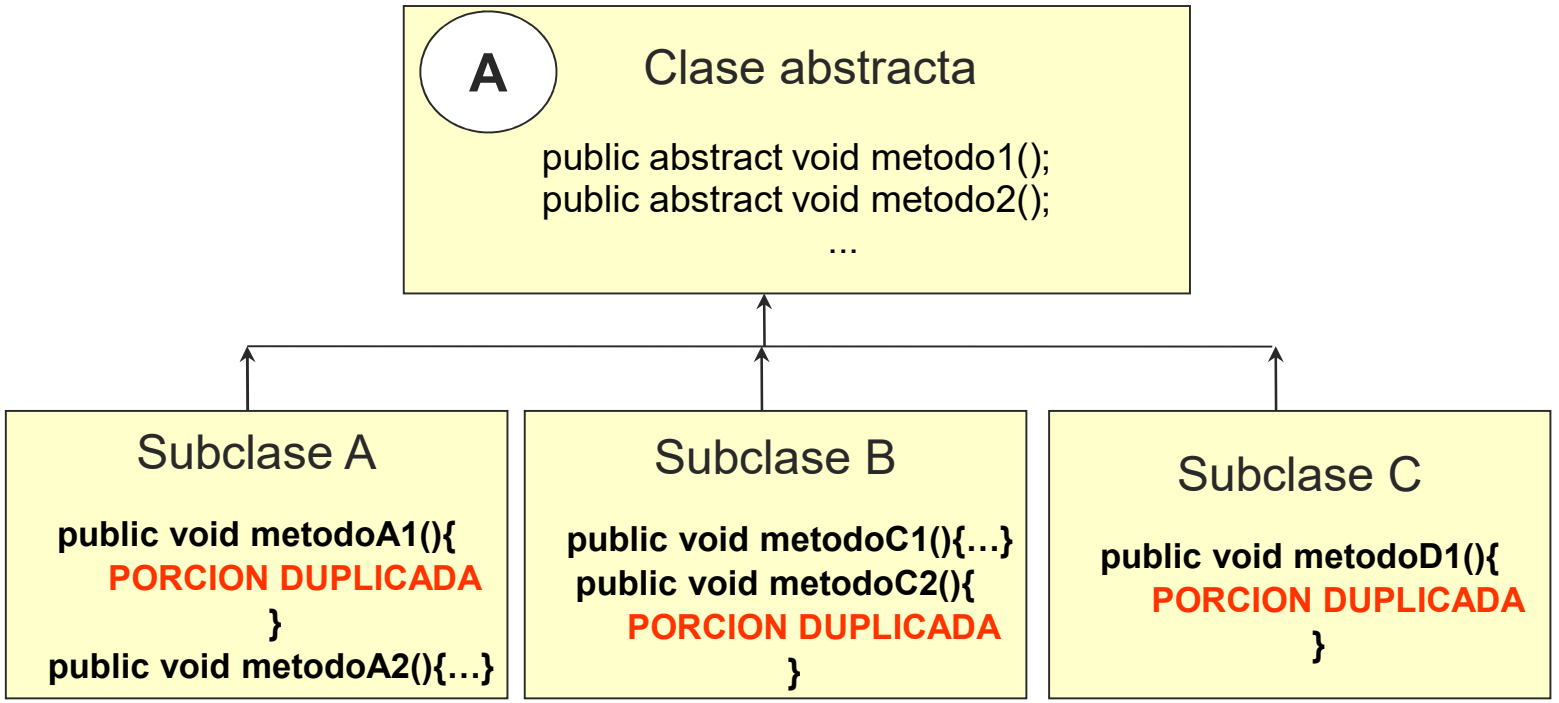
Clases abstractas y duplicación de código

- Ejemplo mejorado: Se han **unificado** las porciones repetidas en un solo lugar, es decir, un método que calcula el promedio basado en un argumento (goles mensuales de un jugador particular)

```
public class DuplicacionTest {
  public static int calcularPromedioAnual(int[] arreglo_goles_jugador){
   int suma = 0;
   for (int i = 0; i < 10; i++)
    suma += arreglo_goles_jugador[i];
   int promedio_anual = suma / 10;
   return promedio_anual;
  }

  public static void main(String[] args) {
   // goles por mes anotados por dos delanteros
   int[] arreglo_goles_jugadorA = new int[10];
   int[] arreglo_goles_jugadorB = new int[10];

   // Las dos porciones duplicadas estaban acá...

   int promedio_anualA = calcularPromedioAnual(arreglo_goles_jugadorA);
   int promedio_anualB = calcularPromedioAnual(arreglo_goles_jugadorB);

   System.out.println(promedio_anualA);
   System.out.println(promedio_anualB);
  }
}
```

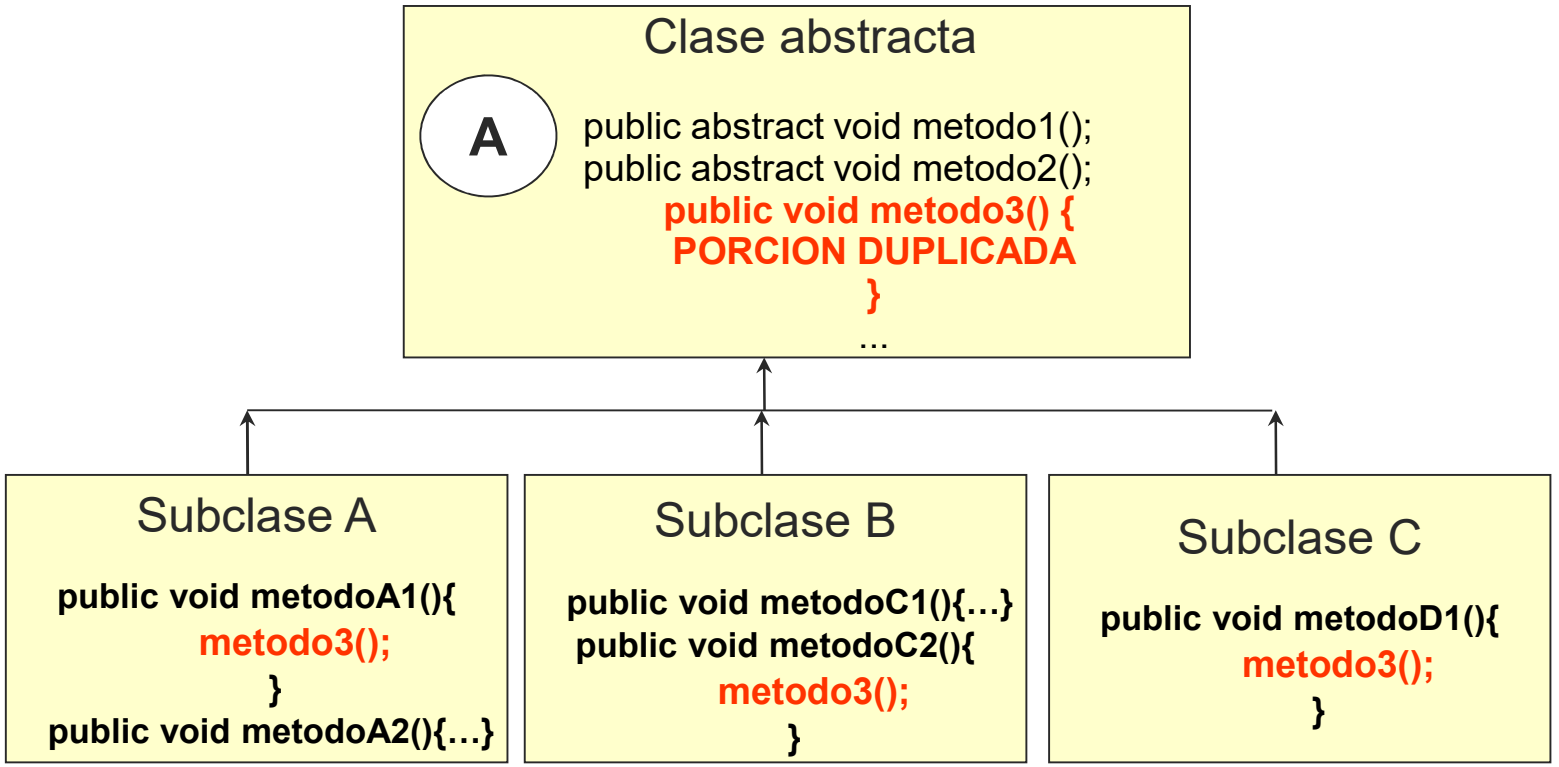
Clases abstractas y duplicación de código

- En programación orientada a objetos, las **clases abstractas** nos permiten proveer una **unica** implementación para un método que es heredado por todas las subclases → evito implementar el mismo método en todas las subclases y **duplicación de código**



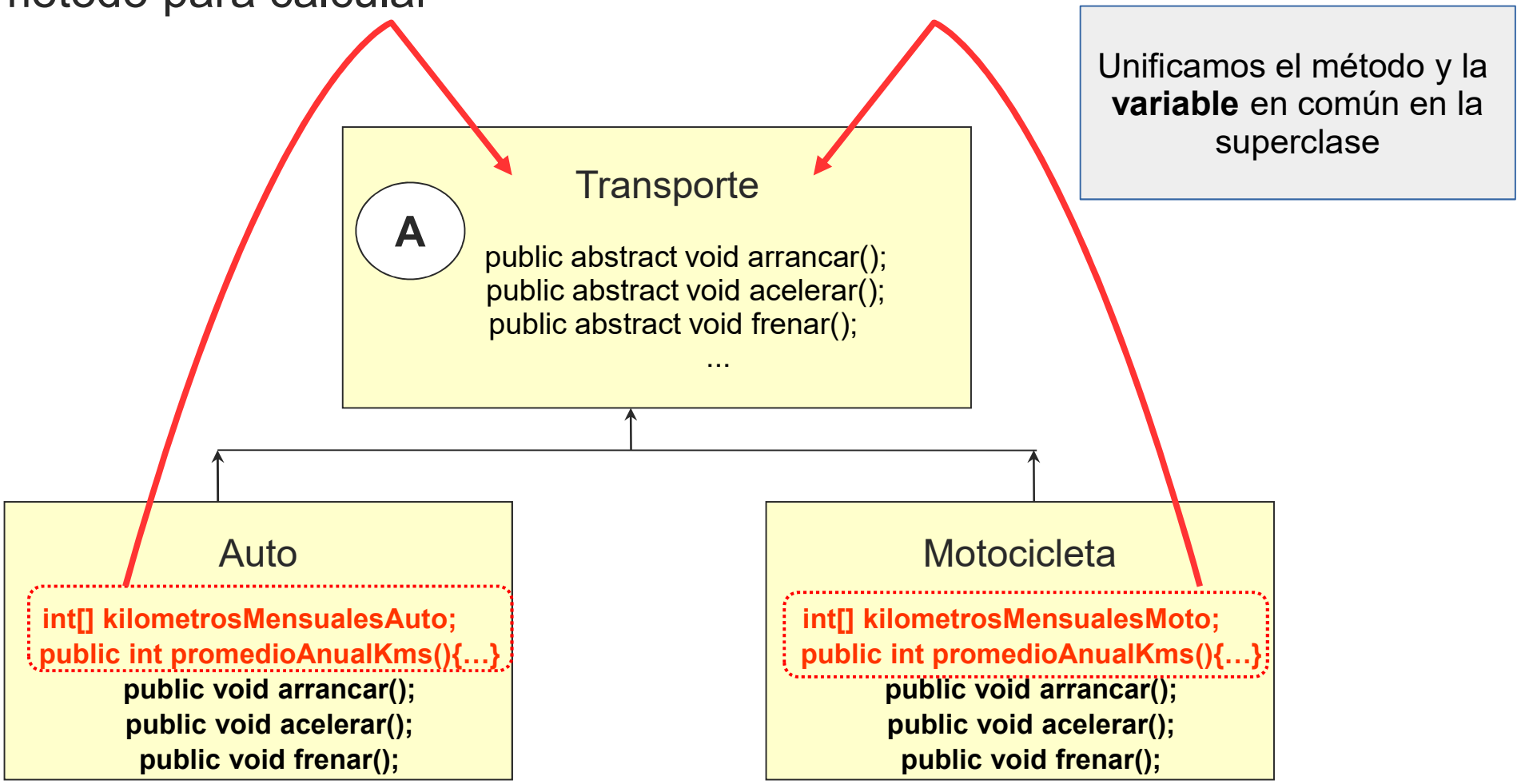
Clases abstractas y duplicación de código

- La porción duplicada de código ahora se ha implementado como un **método concreto** en la superclase
- Las subclases han reemplazado las porciones duplicadas por un llamado al método, análogamente al ejemplo de los jugadores de fútbol



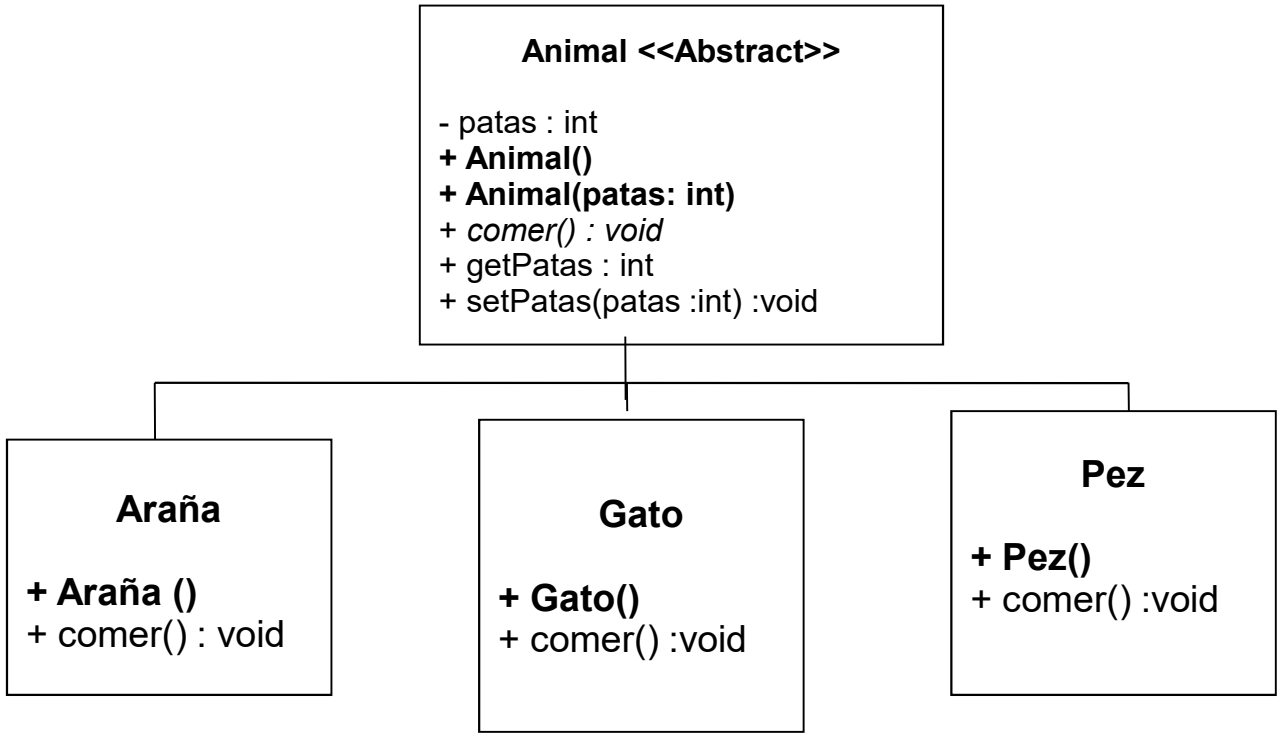
Clases abstractas y duplicación de código

- Ejemplo instanciado: dada la jerarquía de clases que implementan distintos tipos de transporte, podríamos tener en cada transporte un método para calcular



Ejercicios en PC – parte 1

- Codifique la jerarquía de clases descripta en el diagrama UML a continuación en un paquete llamado **modelo**. Todas las clases son públicas. Considere que la implementación del método **comer()** en las distintas subclases de **Animal** consiste en imprimir en pantalla el alimento que ingiere el animal. Por ejemplo, en la clase **Araña** imprime “La araña come insectos”. Los métodos **getPatas()** y **setPatas()** son concretos



Ejercicios en PC – parte 2

Cree una clase llamada **TestAnimales** en un paquete llamado **test**, con las siguientes características:

- En su método main() crea una instancia de Araña, otra de Gato y una de Pez
- Declare un método de clase llamado **muestraQueCome(Animal a)** que recibe como argumento un objeto de tipo Animal y devuelve void. Este método simplemente invoca al método comer() del objeto que recibe como argumento
- Desde el método main() invoque al método muestraQueCome(Animal a) enviándole como argumento cada una de las instancias creadas en el main()
- ¿Qué métodos comer() fueron invocados en cada caso? ¿Se aplicó la idea de polimorfismo?

Ejercicios en PC – parte 3

- Volviendo a la jerarquía de clases inicial, ¿considera correcto que la clase **Animal** establezca que todos los animales tienen **patas**? Si es así, ¿cómo podría corregirse la jerarquía? Implemente entonces la nueva jerarquía modificada, y revise la clase **TestAnimales** para introducir también las adaptaciones necesarias