Abstracción, Encapsulamiento, Herencia y Polimorfismo

75.07 / 95.02 ALGORITMOS Y PROGRAMACIÓN III

Elaboración: Lihuén Carranza y Luca Salluzzi
(en base a las presentaciones de Prof. Dr. Diego Corsi)

Índice

1. Introducción

Los cuatro pilares de la Programación Orientada a Objetos

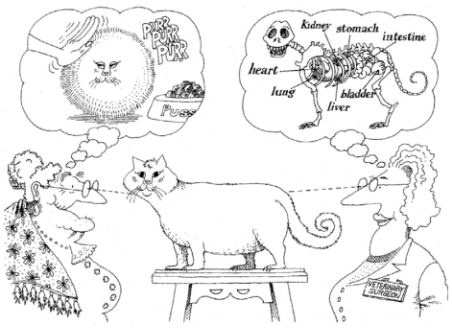
$$\textit{Lenguajes orientados a objetos}^{1} \begin{cases} \textit{Lenguajes basados en objetos} \begin{cases} \textit{Abstraccin} \\ \textit{Encapsulamiento} \end{cases} \\ \textit{Herencia} \\ \textit{Polimorfsmo} \end{cases}$$

Estas cuatro caracteristicas no son inherentes del lenguaje, son un estilo de programación. Uno tecnicamente podría tomar un lenguaje que no sea orientado a objetos (por ejemplo C), y utilizar estos cuatro fundamentos. Los lenguajes orientados a objetos son los que ofrecen estos 4 pilares de forma nativa. Los lenguajes basados en objetos son los que traen de base Abstracción y Encapsulamiento (por ejemplo, Ada en su salida), mientras que los lenguajes orientados a objetos son aquellos que traen de base los 4 pilares.

2. Abstracción

La abstracción es una de las formas fundamentales en que nosotros, como humanos, enfrentamos la **complejidad**. Surge del reconocimiento de **similitudes** entre ciertos objetos del mundo real, y la decisión de concentrarse en estas similitudes e ignorar por el momento las **diferencias**.

Una buena abstracción **enfatiza** los **detalles que son significativos** para el usuario y **suprime** los detalles que son, al menos por el momento, **irrelevantes**.



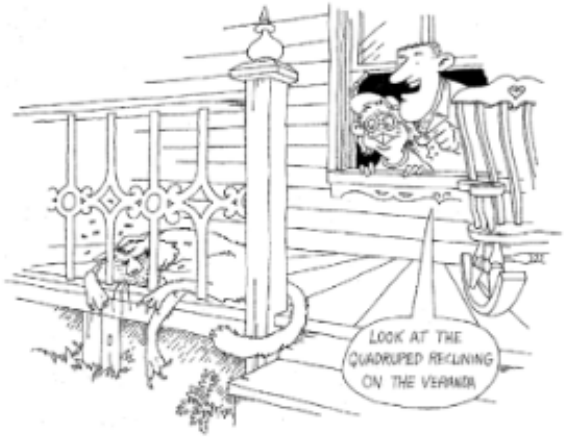
Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

2.1. Principios de la Abstracción

Una abstracción se enfoca en la vista exterior de un objeto y sirve para separar su comportamiento esencial de su implementación.

- **Principio de mínimo compromiso (Principle of least commitment)**: la interfaz de un objeto proporciona su comportamiento esencial, y nada más.
- **Principio del menor asombro (Principle of least astonishment)**: una abstracción captura todo el comportamiento de algún objeto, ni más ni menos, y no ofrece sorpresas ni efectos secundarios que vayan más allá del alcance de la abstracción.

[1] Fuente: Booch, G. et al. (2007). Object-Oriented Analysis and Design with Applications, 3ra. ed., p. 538



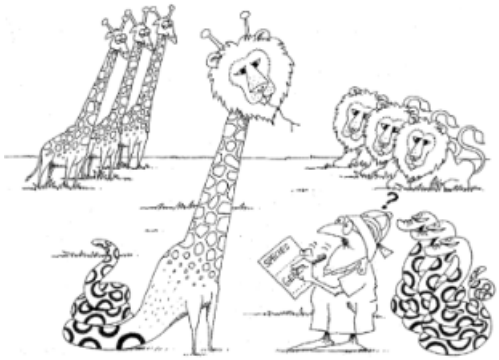
Classes and objects should be at the right level of abstraction: neither too high nor too low.

3. Herencia

En el diseño orientado a objetos, reconocer la **semejanza** entre las cosas nos permite exponer los puntos en común dentro de abstracciones. La **identificación de clases y objetos** implica tanto el descubrimiento como la invención.

A través del **descubrimiento**, llegamos a reconocer las abstracciones que forman el *vocabulario del dominio del problema*.

A través de la **invención**, ideamos abstracciones generalizadas.



Classification is the means whereby we order knowledge.

3.1. La Herencia y la Reutilización de Software

La **herencia** es una forma de reutilización de software en la que se crea una nueva clase aprovechando los miembros de *una* clase existente (*herencia simple*, como en Java) o de varias (*herencia múltiple*, como en C++).

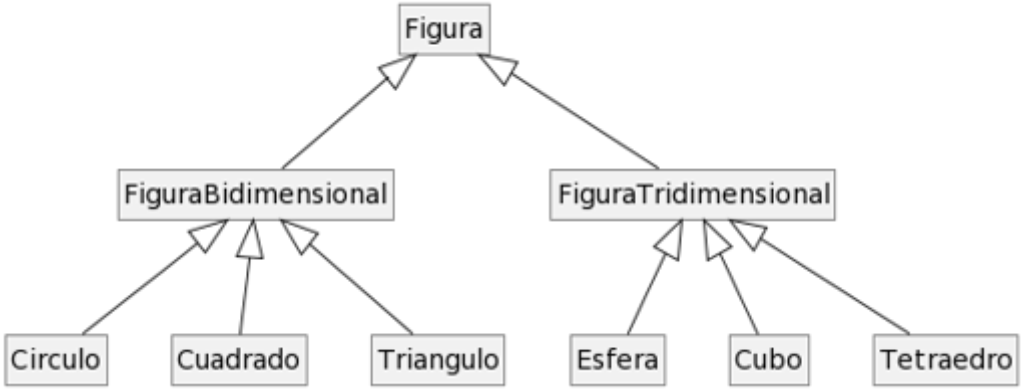
A la clase previamente existente se la conoce como **superclase** (o clase base), y a las nuevas como **subclases** (o *clases derivadas*). Cada subclase se puede convertir en la superclase de futuras subclases.

A subclass may inherit the structure and behavior of its superclass.

3.2. La Herencia y las Jerarquías de Clases

Una subclase generalmente agrega sus propios atributos y métodos. Por lo tanto, una subclase es más específica que su superclase y representa a un grupo más especializado de objetos.

Generalmente, la subclase exhibe los comportamientos de su superclase junto con otros adicionales específicos de esta subclase. Es por ello que a la herencia se la conoce a veces como **especialización** o **generalización**: la flecha se lee es un(a).

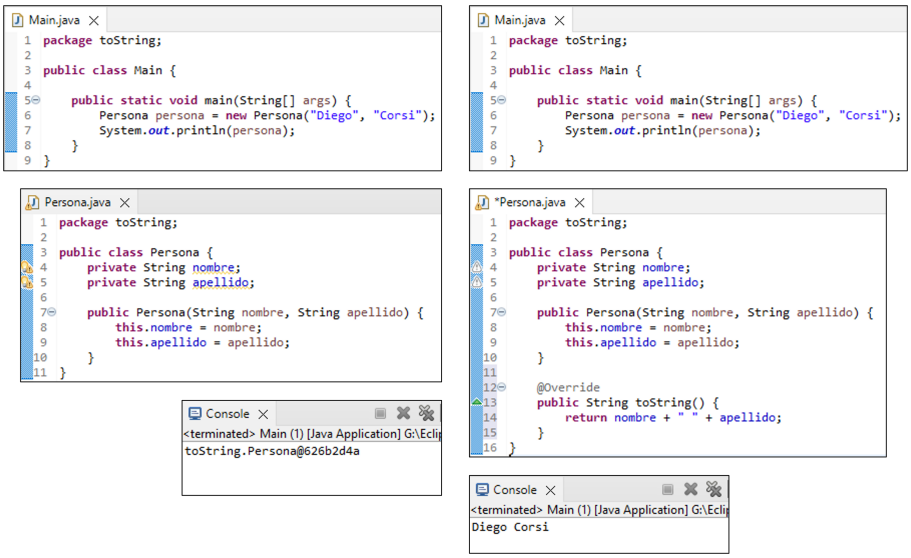


La **superclase directa** es la clase de la cual la subclase hereda en forma explícita.

Una **superclase indirecta** es cualquier clase que se encuentre arriba de la superclase directa en la jerarquía de clases, que es la jerarquía que define las relaciones de herencia entre las clases.

3.3. Ejemplo: La Clase Object

En Java, la jerarquía de clases tiene la clase Object (en el paquete java.lang) en su cima. De ella heredan todas las clases en Java, ya sea en forma directa o indirecta Por ejemplo, todas heredan de Object el método toString que es invocado, por ejemplo, al imprimir el objeto.



https://github.com/dcorsi/algo3/tree/main/toStringSinJUnit

3.4. Redefinición de Metodos

Cabe destacar entonces que, a menudo, un método de la superclase puede no ser apropiado para una subclase, ya que en esta se requiere una versión personalizada del método. En dichos casos,

la subclase puede **sobrescribir** (*redefinir*) el método de la superclase con una implementación apropiada (*method overriding*).

El ejemplo anterior mostró un caso de herencia implícita, ya que todas las clases heredan de Object. La herencia, en Java, se hace explícita mediante la palabra reservada extends. Por lo tanto, aunque resulte redundante, podría haberse declarado la clase Persona de la siguiente manera:

```
public class Persona extends Object {
```

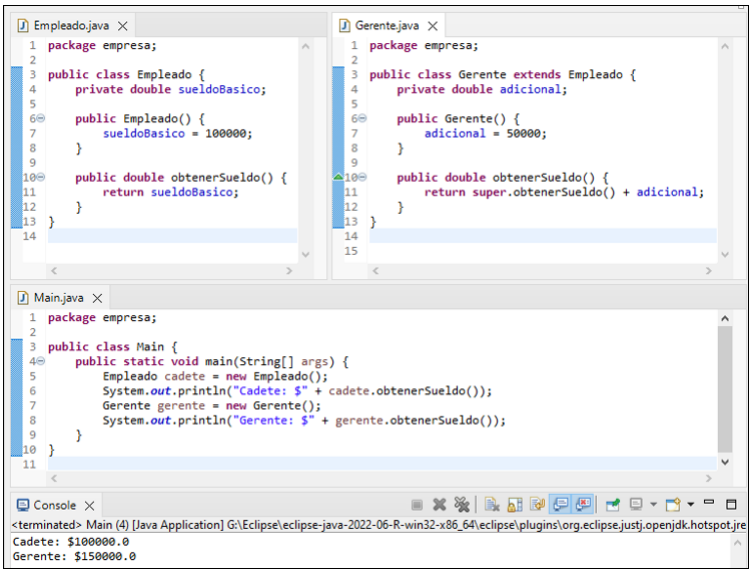
3.5. La Herencia y los Constructores

Los constructores no se heredan. En Java, todas las clases tienen un constructor **por defecto**: el constructor sin parámetros. Igual que con todos los constructores, su primera tarea es llamar al constructor de su superclase directa, para asegurar que las variables de instancia heredadas de la superclase se inicialicen. Esto es así, porque siempre que se crea un objeto de una subclase se empieza una **cadena de llamadas** a los constructores, donde el constructor de la subclase, antes de realizar sus propias tareas, invoca al constructor de la superclase, ya sea en forma explícita (por medio de super) o implícita (llamando al constructor por defecto de la superclase). De igual forma, si la superclase deriva de otra clase, el constructor de la superclase invoca al constructor de la siguiente clase más arriba en la jerarquía, y así sucesivamente.

Si se declara un constructor con parámetros, el constructor por defecto dej de estar disponible y, para poder instanciar objetos sin pasar argumentos, es necesario declarar un nuevo constructor sin parámetros.

3.6. Herencia y Super

Cuando un método de una subclase sobrescribe un método de una superclase, se puede acceder al método de la superclase desde la subclase, si se antepone al nombre del método la palabra clave super y un separador punto (.)



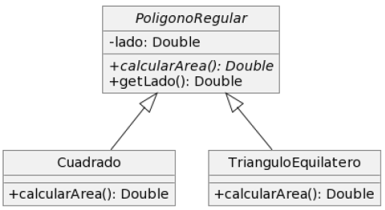
https://github.com/dcorsi/algo3/tree/main/empresa

3.7. La Herencia y las Clases Abstractas

A veces, todo o parte del comportamiento de ciertos objetos es demasiado específico como para implementarlo en una superclase. En tales casos, el método se debe declarar anteponiéndole la palabra reservada abstract en la superclase e implementarse en cada subclase. Así, por

contener al menos un método abstracto, también la superclase debe definirse anteponiéndole la palabra reservada abstract. En consecuencia, ya no será posible crear objetos de esa clase, porque las clases abstractas no son instanciables. El siguiente ejemplo muestra cómo en la clase abstracta PoligonoRegular se declara el método abstracto calcularArea, el cual se implementa luego en las clases TrianguloEquilatero y Cuadrado, dos subclases de PoligonoRegular.

El siguiente ejemplo muestra cómo en la clase abstracta abstracta `PoligonoRegular` se declara el método abstracto `calcularAre`, el cual se implementa luego en las clases `TrianguloEquilatero` y `Cuadrado`, dos subclases de `PoligonoRegular`.

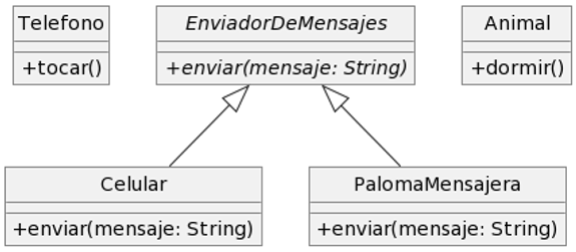


3.8. Limitación de la Herencia Múltiple

La herencia simple soluciona de raíz el problema básico de la herencia múltiple: el problema del diamante (llamado así por la forma que tiene el diagrama de clases donde ocurre este problema). Supongamos que la clase Vehiculo define un método avanzar. Las clases Auto y Lancha extienden Vehiculo: un auto es un vehículo y Ahora bien, dado que un vehículo anfibio es un auto y también es una lancha, podría declararse mediante herencia múltiple que VehiculoAnfibio extiende Auto y Lancha. ¿Qué comportamiento debería tener entonces un vehículo anfibio: el método avanzar que hereda de Auto o el que hereda de Lancha?

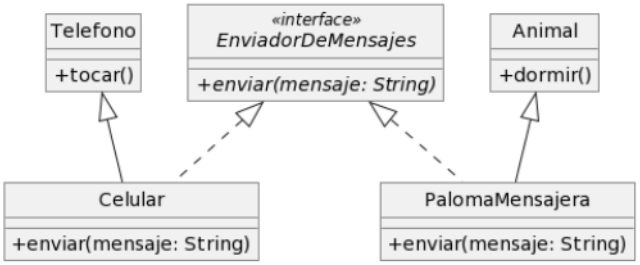
3.9. Limitaciones de la Herencia Simple

La inexistencia de la herencia múltiple podría resultar limitante para ciertos diseños. Por ejemplo, ¿qué tienen en común un celular y una paloma mensajera? La respuesta es que ambos permiten enviar mensajes, por eso podrían modelarse mediante Celular y PalomaMensajera, dos subclases de la clase abstracta EnviadorDeMensajes que deberían implementar el método abstracto enviar declarado en esa clase. Sin embargo, este diseño es muy limitado, ya que la herencia simple impide que Celular herede también de Telefono, o que PalomaMensajera herede de Animal, dos clases llenas de funcionalidad.



3.10. Herencia (Generalización) vs. Realización

La solución consiste modelar como realizaciones todas las relaciones con EnviadorDeMensajes, que no sería una clase sino una interfaz (interface) implementada por Celular (que extiende Telefono) y por PalomaMensajera (que extiende Animal). Haciendo que, en las realizaciones, los métodos de las interfaces carezcan de implementaciones, se evita el problema del diamante.



Main.java

```
package comunicaciones;

public class Main {

    public static void main(String[] args) {
        Celular c = new Celular();
        c.tocar();
        c.enviar("Hola");
        PalomaMensajera p = new PalomaMensajera();
        p.enviar("Hola");
        p.dormir();
    }
}
```

EnviadorDeMensajes.java

```
package comunicaciones;

public interface EnviadorDeMensajes {

    public void enviar(String mensaje);
}
```

Telefono.java

```
package comunicaciones;

public class Telefono {
    public void tocar() {
        System.out.println("RING RING RING");
    }
}
```

Celular.java

```
package comunicaciones;

public class Celular extends Telefono implements EnviadorDeMensajes {

    public void enviar(String mensaje) {
        System.out.println("RING RING: " + mensaje);
    }
}
```

Animal.java

```
package comunicaciones;

public class Animal {

    public void dormir() {
        System.out.println("ZZZ...");
    }
}
```

PalomaMensajera.java

```
package comunicaciones;

public class PalomaMensajera extends Animal implements EnviadorDeMensajes {

    public void enviar(String mensaje) {
        System.out.println("Llega volando: " + mensaje);
    }
}
```

Console

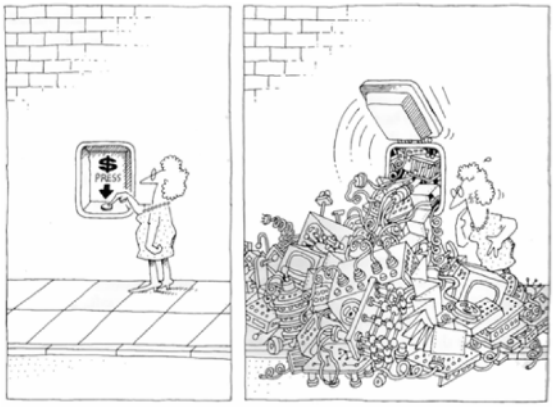
```
<terminated> Main (6) [Java Application] G:\Eclipse\eclipse-java-2022-06-
RING RING RING
RING RING: Hola
Llega volando: Hola
ZZZ...
```

https://github.com/dcorsi/algo3/tree/main/comunicaciones

4. Encapsulamiento

El término *encapsulamiento* suele utilizarse para referirse a:

1. la **agrupación de estado y comportamiento** en una unidad conceptual (la clase);
2. la aplicación de mecanismos de *restricción de acceso a los atributos y métodos*;
3. el **ocultamiento de información** como principio de diseño

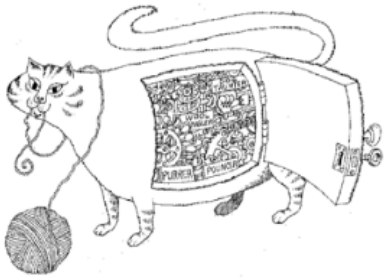


The task of the software development team is to engineer the illusion of simplicity.

El primer significado de *encapsulamiento* refleja el hecho de que los objetos son entidades que agrupan los atributos y los métodos que operan sobre ellos. Así, **un objeto es una cápsula** cuyo interior (su *implementación*) sólo se debería poder utilizar a través de puntos de contacto con el exterior (su *interfaz*) bien definidos.

El segundo significado se refiere al control de la **accesibilidad** o visibilidad de los métodos y atributos desde el exterior, mediante palabras claves (*especificadores de acceso*), que definen qué miembros son la interfaz y cuáles la implementación.

Finalmente, el **ocultamiento de información** es un principio que sugiere que, para obtener un buen diseño, todos los detalles de la implementación deben ser invisibles desde el exterior.



Encapsulation hides the details of the implementation of an object.

En Java, la declaración de una clase de objetos garantiza la primera acepción de encapsulamiento. Sin embargo, esto no garantiza de ningún modo el ocultamiento de la información (tercera acepción). Este se consigue manteniendo los **atributos privados** y que el acceso a los mismos se lleve a cabo mediante **métodos públicos** (segunda acepción).

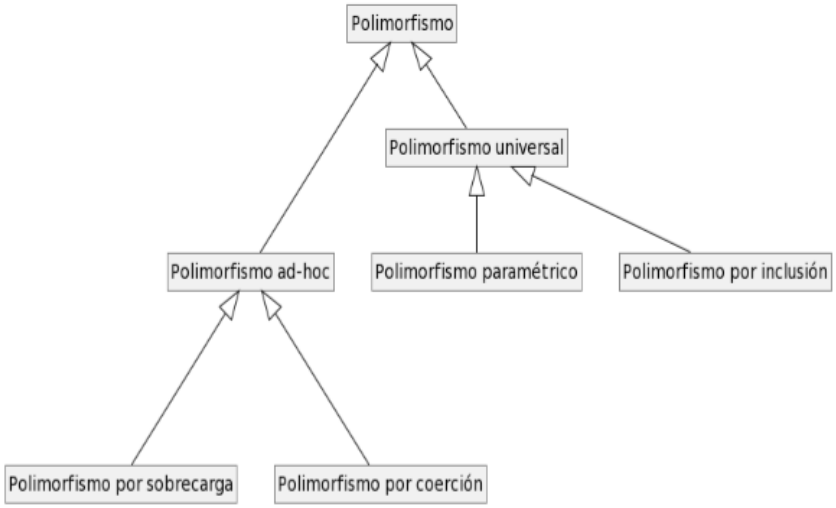
Java ofrece **cuatro niveles de accesibilidad** para los miembros de una clase: visibilidad de paquete (por defecto) y la que se obtiene al anteponerle a cada declaración alguno de los especificadores de acceso `public, private o protected`.

5. Polimorfismo

En griego, $\pi o \lambda \lambda \upsilon \varsigma \ \mu o \phi \eta$ significa "muchas formas". Es por ello que, en la teoría de los lenguajes de programación, se denomina así a la capacidad de contener valores de distintos tipos (en el caso de las *variables polimórficas*) o a la capacidad de recibir argumentos de diferentes tipos (en el caso de los *métodos polimórficos*).

Different observers will classify the same object in different ways.

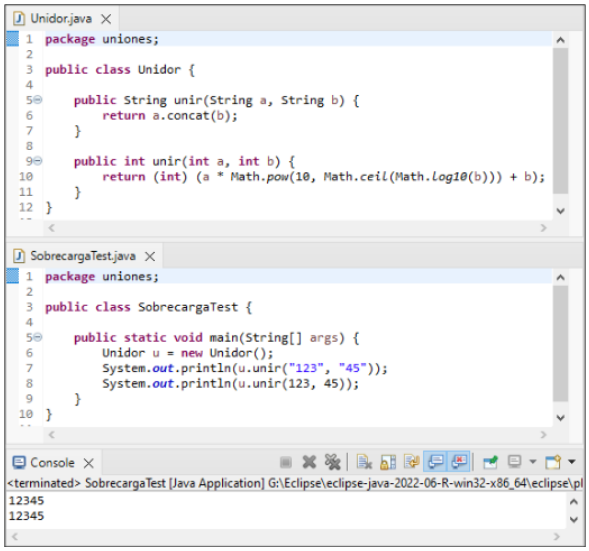
Según cómo sea el conjunto de los tipos posibles, Cardelli y Wegner clasifican el polimorfismo en dos categorías: **polimorfismo ad-hoc** (funciona con un número limitado y conocido de tipos, no necesariamente relacionados entre sí) y **polimorfismo universal** (funciona con un número prácticamente infinito de tipos relacionados entre sí). [2]



5.1. Polimorfismo por sobrecarga

Dentro de una misma clase, es posible escribir dos o más métodos con el mismo nombre pero diferente firma (*signature*). El resultado es un polimorfismo *aparente*, ya que no se trata de un método que puede recibir muchos tipos de argumentos, sino que hay varios métodos con el mismo nombre, y durante el proceso de *compilación* se elije cuál usar según el o los argumento(s) pasado(s).

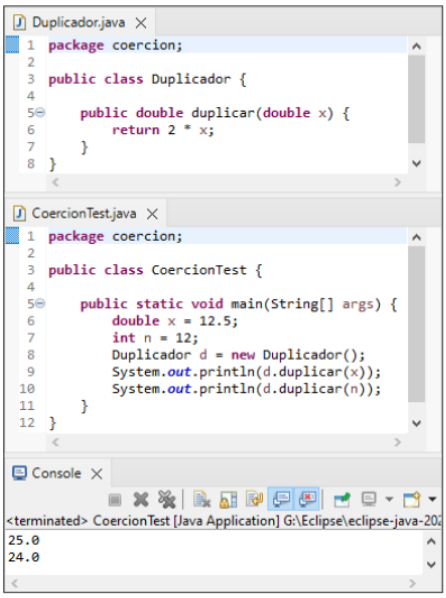
[2] Fuente: Cardelli, L. Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism". En: Computing Surveys (Diciembre, 1985). Vol. 17, n. 4, p. 471



https://github.com/dcorsi/algo3/tree/main/uniones

5.2. Polimorfismo por coerción

Aplicar *coerción* sobre alguien significa forzar su voluntad o su conducta. En programación, significa forzar a que un dato de un tipo sea tratado como si fuera de otro tipo. Por ejemplo, coerción es la operación realizada para convertir (de manera implícita) un argumento al tipo del parámetro correspondiente. En este caso, también se trata de un polimorfismo *aparente*, ya que la conversión que ocurre no cambia el hecho de que el método, en realidad, trabaja con un único tipo.

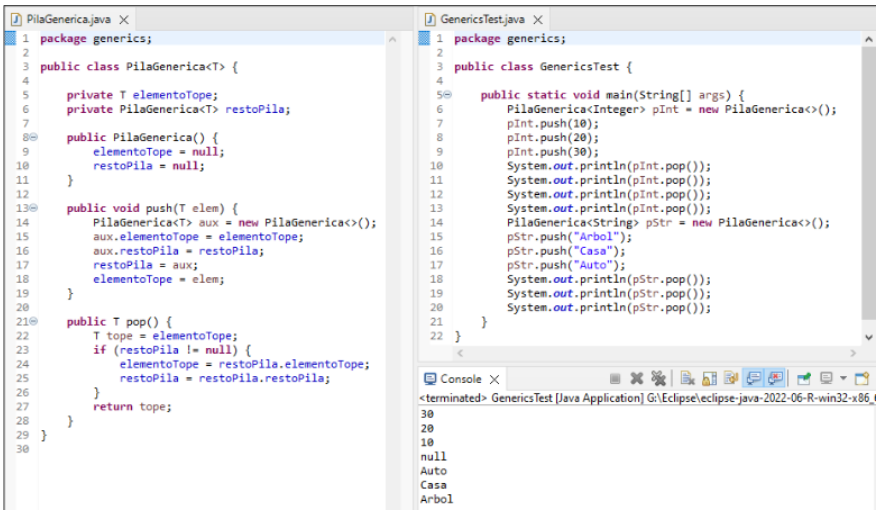


https://github.com/dcorsi/algo3/tree/main/coercion

5.3. Polimorfismo por paramétrico

Cuando se escribe *código genérico*, es decir, sin mencio- nar ningún tipo de datos es- pecífico, para que pueda ser usado con datos que serán tratados de manera idéntica independientemente de su tipo, se está aprovechando el *polimorfismo paramétrico*. En Java, a esta variante de polimorfismo

se la conoce como *Generics*. En el siguien- te ejemplo, la clase textttPilaGenerica puede usarse para instanciar pilas específicas para cualquier tipo de objeto.



```
// PilaGenerica.java
package generics;

public class PilaGenerica<T> {

    private T elementoTope;
    private PilaGenerica<T> restoPila;

    public PilaGenerica() {
        elementoTope = null;
        restoPila = null;
    }

    public void push(T elem) {
        PilaGenerica<T> aux = new PilaGenerica<>();
        aux.elementoTope = elementoTope;
        aux.restoPila = restoPila;
        restoPila = aux;
        elementoTope = elem;
    }

    public T pop() {
        T tope = elementoTope;
        if (restoPila != null) {
            elementoTope = restoPila.elementoTope;
            restoPila = restoPila.restoPila;
        }
        return tope;
    }
}
```

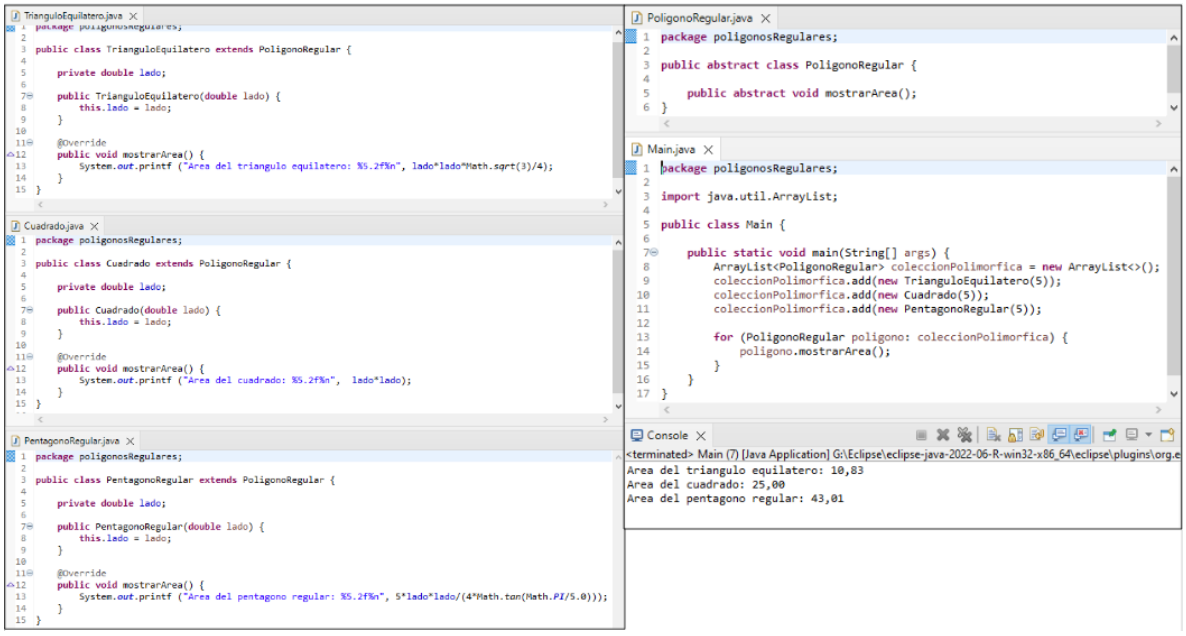
```
// GenericsTest.java
package generics;

public class GenericsTest {

    public static void main(String[] args) {
        PilaGenerica<Integer> pInt = new PilaGenerica<>();
        pInt.push(10);
        pInt.push(20);
        pInt.push(30);
        System.out.println(pInt.pop());
        System.out.println(pInt.pop());
        System.out.println(pInt.pop());
        System.out.println(pInt.pop());
        PilaGenerica<String> pStr = new PilaGenerica<>();
        pStr.push("Arbol");
        pStr.push("Casa");
        pStr.push("Auto");
        System.out.println(pStr.pop());
        System.out.println(pStr.pop());
        System.out.println(pStr.pop());
    }
}
```

```
Console
<terminated> GenericsTest [Java Application] G:\Eclipse\eclipse-java-2022-06-R-win32-x86_
30
20
10
null
Auto
Casa
Arbol
```

https://github.com/dcorsi/algo3/tree/main/generics

5.4. Polimorfismo por inclusión

El *polimorfismo por inclusión*, también conocido como *polimorfismo por herencia* o *polimorfismo de subclases* (o de *subtipos*), es la variante más común encontrada en la POO y, por ello, la mayoría de las veces se lo denomina *polimorfismo* a secas. Se basa en el hecho de que las superclases *incluyen* a todas las instancias de sus subclases o, dicho de otra forma, todos los objetos de una subclase son también objetos de las superclases de esta. Por ello, aunque se declare un atri- buto, una variable local, un parámetro o el contenido de una colección como siendo de una superclase, en realidad puede referirse a cualquier instancia de esa clase o de alguna sus subclases. Los objetos no pierden sus características (estado y com-portamiento) al ser referenciados de esta manera, por lo que es posible invocar los métodos que estos hayan sobrescrito, y su comportamiento será el



```
// TrianguloEquilatero.java
public class TrianguloEquilatero extends PoligonoRegular {

    private double lado;

    public TrianguloEquilatero(double lado) {
        this.lado = lado;
    }

    @Override
    public void mostrarArea() {
        System.out.printf ("Area del triangulo equilatero: %5.2f%n", lado*lado*Math.sqrt(3)/4);
    }
}
```

```
// Cuadrado.java
package poligonosRegulares;

public class Cuadrado extends PoligonoRegular {

    private double lado;

    public Cuadrado(double lado) {
        this.lado = lado;
    }

    @Override
    public void mostrarArea() {
        System.out.printf ("Area del cuadrado: %5.2f%n",  lado*lado);
    }
}
```

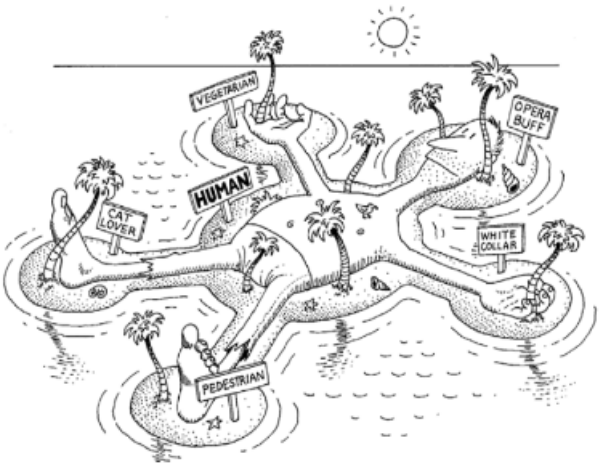
```
// PentagonoRegular.java
package poligonosRegulares;

public class PentagonoRegular extends PoligonoRegular {

    private double lado;

    public PentagonoRegular(double lado) {
        this.lado = lado;
    }

    @Override
    public void mostrarArea() {
        System.out.printf ("Area del pentagono regular: %5.2f%n", 5*lado*lado/(4*Math.tan(Math.PI/5.0)));
    }
}
```

```
// PoligonoRegular.java
package poligonosRegulares;

public abstract class PoligonoRegular {

    public abstract void mostrarArea();
}
```

```
// Main.java
package poligonosRegulares;

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        ArrayList<PoligonoRegular> coleccionPolimorfica = new ArrayList<>();
        coleccionPolimorfica.add(new TrianguloEquilatero(5));
        coleccionPolimorfica.add(new Cuadrado(5));
        coleccionPolimorfica.add(new PentagonoRegular(5));

        for (PoligonoRegular poligono: coleccionPolimorfica) {
            poligono.mostrarArea();
        }
    }
}
```

```
Console
<terminated> Main (7) [Java Application] G:\Eclipse\eclipse-java-2022-06-R-win32-x86_64\eclipse\plugins\org.e
Area del triangulo equilatero: 10,83
Area del cuadrado: 25,00
Area del pentagono regular: 43,01
```

https://github.com/dcorsi/algo3/tree/main/polimorfismo



Objects can play many different roles.