UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 5 de Prácticas

Herencia

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2025-2026

Última modificación:
6 de octubre de 2025

Sesión 5: Herencia

Índice

5.1 Herencia en Python

La **herencia** es uno de los pilares fundamentales de la Programación Orientada a Objetos (POO) en Python. Permite que una clase (denominada **clase hija** o **subclase**) herede comportamientos y características de otra clase (la **clase padre** o **superclase**), permitiendo la reutilización de código y la extensión de funcionalidad sin necesidad de reescribir lo que ya existe en la clase padre.

Herencia de Métodos

Cuando heredamos métodos de una clase padre, tenemos **tres formas principales** de gestionar cómo se comportan esos métodos en la clase hija:

- **Heredar y usar el método tal cual**:
  - La subclase hereda un método definido en la clase padre y lo usa sin hacer modificaciones. El método funciona exactamente como en la superclase, sin necesidad de redefinirlo.
  - Esto es útil cuando el comportamiento del método es adecuado para ambas clases, padre e hija, sin cambios.

```
class Padre:
    def metodo(self):
        print("Método en clase Padre")

class Hija(Padre):
    pass  # No se modifica el método, se hereda tal cual

hija = Hija()
hija.metodo()  # Salida: "Método en clase Padre"
```

- **Extender el método del padre**:
  - La subclase puede añadir funcionalidad extra al método heredado. En este caso, se usa **super()** para llamar al método del padre, ejecutando su lógica, y luego se añade nueva funcionalidad específica de la subclase.
  - Esto es útil cuando se quiere mantener parte del comportamiento original del método del padre, pero también se necesita algo adicional en la subclase.

```
class Padre:
    def metodo(self):
        print("Método en clase Padre")

class Hija(Padre):
    def metodo(self):
        super().metodo()  # Llama al método de la clase Padre
        print("Método extendido en clase Hija")

hija = Hija()
hija.metodo()
# Salida:
# "Método en clase Padre"
# "Método extendido en clase Hija"
```

- **Sobreescribir el método**:
  - La subclase puede cambiar por completo el comportamiento del método heredado. Para ello, simplemente redefine el método con la misma declaración, pero con una nueva implementación que no tiene ninguna relación con la del método original.
  - Esto se utiliza cuando el comportamiento del método del padre no es adecuado o necesario para la subclase.

```
class Padre:
    def metodo(self):
        print("Método en clase Padre")

class Hija(Padre):
    def metodo(self):
        print("Método completamente nuevo en clase Hija")

hija = Hija()
hija.metodo()  # Salida: "Método completamente nuevo en clase Hija"
```

Herencia de Atributos

En cuanto a los atributos de una clase, también existen **tres formas principales** de gestionarlos al aplicar herencia:

- **Heredar y usar los atributos tal cual**:
  - Los atributos definidos en la clase padre son heredados por la subclase y se utilizan sin modificación.
  - Esto es útil cuando las subclases necesitan comportarse de manera similar a la clase padre y los atributos no requieren cambios.

```
class Padre:
    def __init__(self, atributo: str):
        self.atributo = atributo

class Hija(Padre):
    pass

hija = Hija("Atributo en clase Hija")
print(hija.atributo)  # Salida: "Atributo en clase Hija"
```

- **Extender los atributos del padre**:
  - Además de los atributos heredados de la clase padre, podemos definir nuevos atributos en la clase hija.
  - Muy común, pues las clases hijas suelen ser una especialización de la clase padre y, por tanto, tienden a definir propiedades adicionales.

```
class Padre:
    def __init__(self, atributo: str):
        self.atributo = atributo

class Hija(Padre):
    def __init__(self, atributo: str, nuevo: str):
        super().__init__(atributo)
        self.nuevo = nuevo

hija = Hija("Atributo en clase Hija", "Atributo nuevo")
print(hija.atributo)  # Salida: "Atributo en clase Hija"
print(hija.nuevo)     # Salida: "Atributo nuevo"
```

- **Ocultar el atributo del padre**:
  - La subclase puede ocultar un atributo heredado del padre redefiniéndolo con el mismo nombre. En este caso, el atributo de la clase hija **sobrescribe** el de la clase padre, de modo que cualquier referencia a dicho atributo usará la versión de la subclase.
  - Este enfoque se usa cuando el mismo atributo debe tener un significado diferente o un valor específico en la subclase.

```
class Padre:
    def __init__(self):
        self.atributo = "Atributo en clase Padre"

class Hija(Padre):
    def __init__(self):
        super().__init__(atributo)
        self.atributo = "Atributo en clase Hija"  # Oculta el atributo de la clase Padre

hija = Hija()
print(hija.atributo)  # Salida: "Atributo en clase Hija"
```

5.2 Ejemplo de herencia: clases Persona y Libro

Partiendo de la base del código ya presentado en la sesión 4 de prácticas, vamos a extender nuestro proyecto para disponer de dos tipos de libros: libros infantiles y libros científicos. Recordemos que hasta ahora tenemos la clase `Libro`:

```
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list['Persona'] = []  # Lista de personas que han leído el libro
        self._propietario: 'Persona' = propietario  # Relación de agregación: un libro tiene un propietario

    # Método getter para el título
    def get_titulo(self) -> str:
        return self._titulo

    # Método getter para el autor
    def get_autor(self) -> str:
        return self._autor

    # Métodos getter y setter para las páginas
    def get_paginas(self) -> int:
        return self._paginas

    def set_paginas(self, nuevas_paginas: int) -> bool:
        if nuevas_paginas > 0:
            self._paginas = nuevas_paginas
            return True
        return False

    # Método getter/setter para los lectores del libro
    def get_listado_lectores(self) -> list['Persona']:
        return self._listado_lectores

    def agregar_lector(self, persona: 'Persona') -> None:
        self.get_listado_lectores().append(persona)

    # Métodos getter y setter para el propietario del libro
    def get_propietario(self) -> 'Persona':
        return self._propietario

    def _set_propietario(self, propietario: 'Persona'):
        self._propietario = propietario

    def comprar_libro(self, propietario: 'Persona'):
        self._set_propietario(propietario)

    # Método mágico __str__ para obtener la información del libro
    def __str__(self) -> str:
        if len(self.get_listado_lectores()) > 0:
            lectores: str = ", ".join([persona.get_nombre() for persona in self.get_listado_lectores()])
        else:
            lectores: str = "Nadie lo ha leído"

        if self.get_propietario():
            propietario: str = self.get_propietario().get_nombre()
        else:
            propietario: str = "Sin propietario"

        return (f"'{self.get_titulo()}' de {self.get_autor()}, {self.get_paginas()} páginas. "
                f"Leído por: {lectores}")
```

Los libros infantiles se caracterizan por estar dirigidos a niños, por lo que añaden nuevas propiedades sobre los libros comunes: tienen una edad recomendada de lectura, pueden tener o no ilustraciones, y pueden ser o no interactivos. En base a ello, presentamos una primera versión del código de la clase LibroInfantil:

```
class LibroInfantil(Libro):

    def __init__(self, titulo: str, autor: str, paginas: int, edad_recomendada: int, ilustraciones: bool,
                 interactividad: bool, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario) # Llamada al constructor de la clase padre

        # Nuevos atributos de la clase hija
        self._edad_recomendada = edad_recomendada # Edad recomendada de lectura
        self._ilustraciones = ilustraciones # El libro tiene ilustraciones
        self._interactividad = interactividad # El libro es interactivo

    # (...) Métodos getter/setter obviados por simplicidad: métodos nuevos sobre la clase padre

    # Ejemplo de extensión de funcionalidad heredada
    def __str__(self) -> str:
        # Obtiene la descripción básica del libro (superclase)
        descripcion_basica: str = super().__str__()
```

```
        # Detalles extra por ser libro infantil
        detalles_infantiles: str = (f"Edad recomendada: {self.get_edad_recomendada()}, "
                                    f"Ilustraciones: {'Sí' if self.get_ilustrado() else 'No'}, "
                                    f"Interactividad: {'Sí' if self.get_interactivo() else 'No'}")
        return f"{descripcion_basica} | {detalles_infantiles}"

    '''
    Ejemplo de extensión de funcionalidad:
        - Se añade funcionalidad al método de la clase hija: comprobación de la edad
        - Se hace uso de super().agregar_lector() para llamar al método de la clase padre
    '''
    def agregar_lector(self, persona: 'Persona') -> None:
        # get_edad() viene heredado de la superclase
        if persona.get_edad() < self.get_edad_recomendada():
            print(f"{persona.get_nombre()} es demasiado joven para leer este libro infantil.")
        else:
            # Si la edad es adecuada, llamamos al método de la clase padre
            super().agregar_lector(persona)
            print(f"{persona.get_nombre()} ha sido añadido como lector del libro '{self.get_titulo()}'.")
```

En relación a los atributos, se definen simplemente tres nuevos atributos que extienden la definición dada por la clase Libro, no realizando en ningún caso ocultación de los atributos de la clase padre. En cuanto a la herencia de métodos podemos ver cómo *LibroInfantil* hereda todos los métodos definidos en la clase Libro, tales como los getter/setter y el método *comprar_libro()*. En estos casos, se realizará un uso directo de los métodos definidos en la clase Libro, no teniendo que realizar ningún cambio en la clase *LibroInfantil*. Además, la clase *LibroInfantil* definiría nuevos métodos getter/setter sobre los atributos adicionales, que no se dejan indicados en el código por brevedad.

Sin embargo, esta clase presenta dos casos de extensión de funcionalidad sobre métodos definidos en la superclase. En primer lugar, el método __str__ define su propia implementación teniendo como base la de la clase Libro. Para ello, hace uso de *super()* sobre el método __str__ de la clase padre y, una vez ha obtenido la descripción básica, pasa a cumplimentar la información específica de los libros infantiles. Una vez terminada, este método ahora se encarga de mostrar la información tanto básica como la específica infantil.

En segundo lugar, se extiende la funcionalidad del método *agregar_lector()* para añadir una restricción de edad. Recordemos que un libro infantil tiene una edad de lectura recomendada, por lo que no debería poder leerse si la persona es menor a dicha recomendación. En el caso de que el lector sea mayor de esa edad, sí podremos agregar el libro, para lo que nos apoyamos del método *agregar_lector()* de la superclase, otra vez haciendo uso de *super()*. Finalmente, en esta clase no hay ningún método que sobrescriba por completo la funcionalidad heredada.

Por su parte, los libros científicos se caracterizan por estar enmarcados en el contexto de un campo de estudio, tienen un determinado nivel de dificultad para comprenderlos y, finalmente, disponen de un listado de referencias a otros libros en los que se apoyan a nivel bibliográfico. Pasemos ahora a revisar el contenido de la clase LibroCientifico:

```
class LibroCientifico(Libro):
    def __init__(self, titulo: str, autor: str, paginas: int, campo_estudio: str, nivel_dificultad: str,
                 propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario)

        # Nuevos atributos de la clase hija
        self._campo_estudio: str = campo_estudio
        self._nivel_dificultad: str = nivel_dificultad
        self._referencias: list['Libro'] = []

    def get_campo_estudio(self) -> str:
        return self._campo_estudio

    def get_nivel_dificultad(self) -> str:
        return self._nivel_dificultad

    def get_referencias(self) -> list['Libro']:
        return self._referencias

    # Ejemplo de nueva funcionalidad añadida en la clase hija
    def agregar_referencia(self, referencia: 'Libro') -> None:
        self.get_referencias().append(referencia)

    # Ejemplo de extensión de funcionalidad heredada
    def __str__(self) -> str:
        descripcion_basica: str = super().__str__()  # Llamada al método __str__ de la clase padre

        # Construcción de los detalles específicos del libro científico
        detalles_cientificos: str = (f"Campo de estudio: {self.get_campo_estudio()}, "
                                     f"Nivel de dificultad: {self.get_nivel_dificultad()}")

        # Añadir las referencias si existen
        if self.get_referencias():
            referencias_str = ", ".join([referencia.get_titulo() for referencia in self.get_referencias()])
            detalles_cientificos += f", Referencias: {referencias_str}"
        else:
            detalles_cientificos += ", No hay referencias"
```

```
        return f"{descripcion_basica} | {detalles_cientificos}"

    '''
    Sobrescritura completa del método:
        - Se añade funcionalidad para verificar si el propietario tiene experiencia previa con libros científicos
        - Se cambia el tipo de retorno, de None a bool
    '''
    def comprar_libro(self, propietario: 'Persona') -> bool:
        # Verificar si la persona ha leído libros científicos con anterioridad.
        # Uso de isinstance() para verificar el tipo de libro
        libros_cientificos_leidos: list['Libro'] = []
        for libro in propietario.get_libros_leidos():
            if isinstance(libro, LibroCientifico):
                libros_cientificos_leidos.append(libro)

        if len(libros_cientificos_leidos) >= 2:  # Tiene experiencia previa leyendo libros científicos
            self._set_propietario(propietario)
            return True
        else:
            return False
```

En este caso también se definen tres nuevos atributos adicionales a los que proporciona la clase padre, sin realizar ocultación de ninguno de ellos. En relación a los métodos, se realiza extensión de funcionalidad en __str__ de forma análoga a lo que se ha explicado previamente en *LibroInfantil*. Además, se añade sobre la clase padre métodos getter/setter sobre los nuevos atributos. Como ejemplo se muestra el método *agregar_referencia()*, que permite añadir nuevas referencias bibliográficas a un libro científico.

Finalmente, el comportamiento del método *comprar_libro()* se sobrescribe por completo. En esta nueva versión, decidimos que solo se podrá comprar un libro científico si el lector tiene experiencia previa en la lectura de este tipo de libros (en el mundo real esto no sucedería, pero nos sirve para ilustrar este tipo de métodos). Así, se recupera el listado de libros leídos de la persona y, si hay al menos dos libros científicos, podrá comprar el libro. Es interesante destacar el uso del método isinstance(), que permite comprobar en tiempo de ejecución si la clase actual corresponde a *LibroCientifico* o a una de sus subclases (en nuestro caso no tiene clases hijas). En el código destaca también el uso del método *_set_propietario()*, que se definió en la clase *Libro* con visibilidad protegida. Dicho nivel de visibilidad nos permite que esté disponible a nivel de las clases hijas, por lo que puede usarse en *LibroCientifico* (y *LibroInfantil*), pero no en clases externas a la relación de herencia, como podría ser la clase *Persona*.

Finalmente, se muestra un ejemplo de *main* en el que se invoca a la funcionalidad indicada previamente:

```
# Crear instancias de Persona
persona1: Persona = Persona("Paco", 9)
persona2: Persona = Persona("Sergio", 25)  # Persona adulta

# Crear un libro base y probar su funcionalidad
libro1: Libro = Libro("El Hobbit", "J.R.R. Tolkien", 300)
persona2.leer_libro(libro1)
print("Libro1: ", libro1)

# Crear un libro infantil y probar su funcionalidad
libro_infantil: LibroInfantil = LibroInfantil("El Principito", "Antoine de Saint-Exupéry", 96, 10, True, True)
print("Libro infantil: ", libro_infantil)

# Agregar lector al libro infantil
libro_infantil.agregar_lector(persona1)  # Edad inadecuada (9 años)
libro_infantil.agregar_lector(persona2)  # Edad adecuada (25 años)

# Crear libros científicos y probar su funcionalidad
libro_cientifico1: LibroCientifico = LibroCientifico("Física Cuántica", "Einstein", 500, "Física", "Avanzado")
libro_cientifico2: LibroCientifico = LibroCientifico("Relatividad General", "Einstein", 300, "Física", "Avanzado")
print("Libro científico 1: ", libro_cientifico1)
print("Libro científico 2: ", libro_cientifico2)

# Probar la compra de los libros científicos según la experiencia de la persona
persona2.leer_libro(libro_cientifico1)  # Sergio lee un libro científico
resultado_compra1: bool = libro_cientifico1.comprar_libro(persona2)  # No tiene suficiente experiencia
print(f"Resultado de la compra del libro científico 1: {resultado_compra1}")  # False

persona2.leer_libro(libro_cientifico2)  # Sergio lee otro libro científico
resultado_compra2: bool = libro_cientifico1.comprar_libro(persona2)  # Ahora puede comprar el libro
print(f"Resultado de la compra del libro científico 1 después de leer más: {resultado_compra2}")  # True

# Ejemplo de agregar una referencia solo en el libro científico
libro_cientifico1.agregar_referencia(libro_cientifico2)

listado_referencias: list[Libro] = libro_cientifico1.get_referencias()

print(f"Referencias en el libro científico 1: ")
for referencia in libro_cientifico1.get_referencias():
    print("> ", referencia)

# Usar type() para comparar tipos exactos
```

```
print("libro_infantil es de tipo LibroInfantil? ", type(libro_infantil) is LibroInfantil) # True
print("libro_cientifico1 es de tipo LibroCientifico? ", type(libro_cientifico1) is LibroCientifico) # True
print("libro_cientifico1 es de tipo Libro? ", type(libro_cientifico1) is Libro) # False

# Usar isinstance() para verificar si un objeto es de una clase o una subclase
print("libro_infantil es instancia de LibroInfantil? ", isinstance(libro_infantil, Libro)) # True
print("libro_cientifico1 es instancia de Libro? ", isinstance(libro_cientifico1, Libro)) # True
print("libro_cientifico1 es instancia de LibroInfantil? ", isinstance(libro_cientifico1, LibroInfantil)) # False

# Usar issubclass() para verificar si una clase es una subclase de otra
print("LibroCientifico es subclase de Libro? ", issubclass(LibroCientifico, Libro)) # True
print("LibroInfantil es subclase de Libro? ", issubclass(LibroInfantil, Libro)) # True
print("LibroInfantil es subclase de LibroCientifico? ", issubclass(LibroInfantil, LibroCientifico)) # False
```

Así, podemos ver que el código crea dos personas y les añade libros, estableciendo las relaciones entre clases de forma adecuada, tal y como vimos en la sesión de prácticas anterior. Así, se define un libro infantil que es leído por ambas personas. Sin embargo, el método *agregar_lector* incluye en el caso de LibroInfantil una restricción en base a la edad de la persona. Por otro lado, en relación a los libros científicos, podemos ver cómo para comprar un libro científico la persona debe haber leído antes al menos dos libros. Finalmente, vemos un ejemplo de uso de *type()*, *isinstance()* y *issubclass()*.

5.3 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 4 donde tenemos implementadas completamente las clases de Polígono, Punto y Línea, incluyendo relaciones entre ellas. A continuación se listan los atributos y métodos que deberían estar definidos en cada clase.

```
Clase Punto

Atributos:
    _coordenada_x: Coordenada X del punto.
    _coordenada_y: Coordenada Y del punto.

Métodos:
    get_coordenada_x()
    set_coordenada_x()
    get_coordenada_y()
    set_coordenada_y()
    get_linea()
    set_linea()
    get_cuadrante()
    calcular_distancia_puntos()
```

```
Clase Linea

Atributos:
    _punto_inicio: Punto inicial de la línea.
    _punto_fin: Punto final de la línea.
    _poligono: Polígono al que pertenece.

Métodos:
    get_punto_inicio()
    set_punto_inicio()
    get_punto_fin()
    set_punto_fin()
    get_poligono()
    set_poligono()
    calcular_longitud()
    mostrar_puntos()
    get_puntos()
```

```
Clase Poligono

Atributos:
    numero_poligonos: Atributo de clase que cuenta el número de polígonos creados.
    distancia_maxima_vecinos: Distancia máxima para determinar vecindad de polígonos.
    _numero_lados: Número de lados del polígono.
    _color: Color del polígono.
    _forma: Forma del polígono
    _relacion_lados: Relación entre los lados
    _lados: Lista de objetos Linea que representan los lados del polígono.
    __contraseña_base_datos: Contraseña interna del polígono.
    _poligonos_vecinos: Listado de polígonos vecinos a un polígono.
    _vertices: Conjunto de vértices asociados a un polígono.

Métodos:
    get_numero_poligonos()
    set_numero_poligonos()
    get_numero_lados()
    set_numero_lados()
    get_color()
    set_color()
    get_forma()
    set_forma()
    get_relacion_lados()
    set_relacion_lados()
    get_lados()
    set_lados()
    eliminar_lado_aleatorio()
    __get_contraseña()
    __set_contraseña()
    cambiar_contraseña()
    get_poligonos_vecinos()
    get_vertices()
    escalar_poligono()
    calcular_perimetro()
    es_vecino()
    agregar_vecino()
    __extraer_vertices()
    __str__()
    __eq__()
    __len__()
    __add__()
    __lt__()
```

Partiendo de la definición de la clase Poligono, vamos a implementar tres nuevos tipos de polígonos: cuadrados, triángulos y círculos. Para ello, veamos cuáles son sus características principales.

1. **Clase Triángulo**.
   - **Atributos**: Un triángulo se caracteriza por ser un polígono de tres lados. Además, un triángulo puede ser de un determinado tipo: escaleno, isósceles o equilátero. Esta propiedad la denominaremos **tipo_triangulo**. *Nota: recuerda que podemos heredar los atributos del padre, extenderlos u ocultarlos*.
   - **Métodos**:
     a) Para el cálculo del perímetro, un triángulo debe verificar que tiene exactamente tres lados. En caso contrario, no se podrá hacer uso de esta funcionalidad.
     b) Un triángulo debe permitir calcular su área de forma conveniente. Para ello, puedes hacer uso de la fórmula de Herón: $A = \sqrt{s(s-a)(s-b)(s-c)}$, donde a, b y c representan las longitudes de cada uno de los tres lados del triángulo, y s representa el semiperímetro, calculado como $s = \frac{a+b+c}{2}$.
     c) No se puede permitir eliminar un lado aleatorio de un triángulo, pues dejaría de ser un polígono. Haz los cambios necesarios para tener en cuenta esta restricción.
     d) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.
   - **Preguntas**:
     a) ¿La clase Triangulo se beneficia de la definición de atributos de la clase Poligono? ¿Tendría sentido entonces definir la relación de herencia?
     b) ¿Tenemos métodos añadidos sobre los definidos en la clase padre? ¿Hay métodos en la superclase que no tengan sentido en la subclase y, por tanto, tengamos que sobreescribirlos?
2. **Clase Círculo**.
   - **Atributos**: Aunque un círculo no es realmente un polígono, en esta sesión lo consideraremos como tal debido a fines didácticos. Así, un círculo tiene como propiedades su **radio** y su **centro**.
   - **Métodos**:
     a) Un círculo debe permitir calcular su área y su perímetro de forma conveniente.
     b) Adapta el método *es_vecinos()* para que los cálculos se realicen en base a la distancia al centro del círculo (entre centros si se trata de dos círculos, o del centro del círculo a cualquier otro vértice del polígono en caso contrario).
     c) El escalado de un círculo consiste en modificar su radio en base a los valores de factor y ajuste, en lugar de su número de lados.
     d) Realiza los cambios necesarios en los métodos para que el código sea compatible con las restricciones con la clase Poligono analizadas el apartado de atributos.
     e) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.
   - **Preguntas**:
     a) ¿La clase Circulo se beneficia de la definición de atributos de la clase Poligono? ¿Tendría sentido entonces definir la relación de herencia?
     b) ¿Tenemos métodos añadidos sobre los definidos en la clase padre? ¿Hay métodos en la superclase que no tengan sentido en la subclase y, por tanto, tengamos que sobreescribirlos?
3. **Clase Cuadrado**.
   - **Atributos**: Un cuadrado se caracteriza por ser un polígono de cuatro lados. Además, definiremos como propiedad de un cuadrado la longitud de su diagonal, denominada **longitud_diagonal**, obtenida a partir de la fórmula $longitud\_lado * sqrt(2)$.
   - **Métodos**:
     a) Para el cálculo del perímetro, un cuadrado debe verificar que tiene exactamente cuatro lados. En caso contrario, no se podrá hacer uso de esta funcionalidad.
     b) Un cuadrado debe permitir calcular su área de forma conveniente.
     c) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.
   - **Preguntas**:
     a) ¿La clase Cuadrado se beneficia de la definición de atributos de la clase Poligono? ¿Tendría sentido

entonces definir la relación de herencia?

b) ¿Tenemos métodos añadidos sobre los definidos en la clase padre? ¿Hay métodos en la superclase que no tengan sentido en la subclase y, por tanto, tengamos que sobreescribirlos?

4. **Definición de nuevas subclases**. Define las clases TrianguloEscaleno, TrianguloIsosceles y TrianguloEquilatero para que podamos instanciar objetos de esos tipos, gestionando la herencia y haciendo las comprobaciones pertinentes para que estas clases sean coherentes. *¿Se extienden nuevos atributos? ¿Se ocultan atributos existentes? ¿Hay sobreescritura completa de métodos o sólo extensiones de los mismos?*