Tipos de datos abstractos

Jorge Mario Londoño Peláez & Varias AI

January 29, 2025

1 Tipos de Datos Abstractos

1.1 Uso de la herencia

La herencia es un concepto fundamental en la Programación Orientada a Objetos (Object Oriented Programming) (POO) que permite crear nuevas clases basadas en clases existentes. Esto promueve la reutilización de código y la creación de jerarquías de clases. En el contexto de los Abstract Data Type (Tipo de Dato Abstracto) (ADT), la herencia se puede utilizar de dos maneras principales: herencia de interfaces y herencia de implementación.

1.1.1 Tipos de herencia

En la POO, la herencia se manifiesta principalmente de dos formas:

- **Herencia de interfaces:** Se enfoca en heredar la estructura (métodos) de una clase abstracta o interfaz, sin heredar la implementación.
- **Herencia de implementación:** Se hereda tanto la estructura como la implementación de una clase base.

1.1.2 La interface como la definición del Application Programming Interface (Interfaz de Programación de Aplicaciones) (API) del ADT

Una interfaz define el API de un ADT, especificando las operaciones que se pueden realizar sobre los datos, sin revelar cómo se implementan estas operaciones. En esencia, una interfaz es un contrato que las clases deben cumplir.

El tipo de una interfaz se puede utilizar como un tipo de datos, lo que permite el polimorfismo. Esto significa que diferentes clases pueden implementar la misma interfaz, y el código cliente puede interactuar con estas clases a través de la interfaz, sin conocer la implementación específica. Esto facilita la creación de sistemas flexibles y extensibles.

1.2 La herencia de implementacion

En algunos casos, es útil heredar la implementación de una clase base, además de su interfaz. Esto permite reutilizar el código de la clase base y evitar la duplicación de código.

En Java, todas las clases heredan implícitamente de la clase `Object`. Esta clase proporciona métodos como `equals`, `hashCode` y `toString`, que son fundamentales para el comportamiento de los ADT.

- `equals`: Permite comparar dos objetos para determinar si son iguales. Es importante sobrescribir este método para que la comparación se base en el contenido del objeto (el valor del ADT), en lugar de comparar sus referencias.
- `toString`: Devuelve una representación en cadena del objeto. Es útil para la depuración y la visualización de los datos. Idealmente todo ADT debe proporcionar la forma de representar textualmente el valor del ADT para por ejemplo visualizarlo fácilmente.
- `hashCode`: Devuelve un código hash para el objeto. Este método es importante para el uso de ADT en estructuras de datos como las tablas asociativas.

1.2.1 Ejemplos

Ejemplo 1: ADT Punto2D Este ejemplo define un ADT `Punto2D` que representa un punto en un plano bidimensional. La interfaz `Punto2D` define los métodos `getX()`, `getY()` y `distancia()`. Se proporcionan dos implementaciones: `Punto2DCartesiano` y `Punto2DPolar`.

```
interface Punto2D {
    double getX();
    double getY();
    double distancia(Punto2D otro);
}
```

Listing 1: Interfaz Punto2D

```
class Punto2DCartesiano implements Punto2D {
    private double x;
    private double y;

    public Punto2DCartesiano(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public double getX() {
        return x;
    }

    @Override
    public double getY() {
        return y;
    }

    @Override
    public double distancia(Punto2D otro) {
        double dx = this.getX() - otro.getX();
        double dy = this.getY() - otro.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Listing 2: Implementación cartesiana de Punto2D

```
class Punto2DPolar implements Punto2D {
    private double radio;
    private double angulo;
```

```
4
5       public Punto2DPolar(double radio, double angulo) {
6           this.radio = radio;
7           this.angulo = angulo;
8       }
9
10
11      @Override
12      public double getX() {
13          return radio * Math.cos(angulo);
14      }
15
16      @Override
17      public double getY() {
18          return radio * Math.sin(angulo);
19      }
20
21     @Override
22      public double distancia(Punto2D otro) {
23          double dx = this.getX() - otro.getX();
24          double dy = this.getY() - otro.getY();
25          return Math.sqrt(dx * dx + dy * dy);
26      }
27  }
```

Listing 3: Implementación polar de Punto2D

Ejemplo 2: Implementación de `equals` Este ejemplo muestra cómo implementar el método `equals` para la clase `Punto2D`. La implementación debe comparar los puntos con una tolerancia para tener en cuenta la precisión limitada de los números de punto flotante.

```
1       @Override
2       public boolean equals(Object obj) {
3           if (this == obj) return true;
4           if (obj == null || !(obj instanceof Punto2D)) return false;
5           Punto2D other = (Punto2D) obj;
6           return distancia(other) < TOL;
7       }
8
9       static final double TOL = 1E-12;
```

Listing 4: Implementación de equals en Punto2D

Ejemplo 3: Implementación de `toString` Este ejemplo muestra cómo implementar el método `toString` para las clases `Punto2DCartesiano` y `Punto2DPolar`.

```
1       @Override
2       public String toString() {
3           return "(" + x + ", " + y + ")";
4       }
```

Listing 5: Implementación de toString en Punto2DCartesiano

```
1       @Override
2       public String toString() {
3          return "(" + radio + "@" + angulo + ")";
4       }
```

Listing 6: Implementación de toString en Punto2DPolar

Ejemplo 4: Prueba unitaria Este ejemplo muestra cómo realizar una prueba unitaria para verificar el correcto funcionamiento del método `equals`.

```
public static void main(String[] args) {
    Punto2D p1 = new Punto2DCartesiano(1, 1);
    Punto2D p2 = new Punto2DCartesiano(1.00001, 1.00001);
    Punto2D p3 = new Punto2DCartesiano(2, 2);
    Punto2D p4 = new Punto2DPolar(Math.sqrt(2), Math.PI / 4);


    assert p1.equals(p2) : "p1 debe ser igual a p2";
    assert !p1.equals(p3) : "p1 no debe ser igual a p3";
    assert p1.equals(p4) : "p1 debe ser igual a p4";

    System.out.println("Pruebas unitarias pasaron");
}
```

Listing 7: Prueba unitaria de equals

1.2.2 Métodos especiales en Python

Los métodos especiales, también conocidos como métodos mágicos o dunder methods (por sus nombres que comienzan y terminan con doble guion bajo), son métodos que permiten personalizar el comportamiento de los objetos en Python. Estos métodos son invocados automáticamente por el intérprete en ciertas situaciones, como cuando se crea un objeto, se imprime, se compara, etc.

En la implementación de Tipo de Dato Abstracto (TDA)s, los métodos especiales son muy útiles para definir cómo se comportan los objetos de nuestro TDA en diferentes contextos.

1.3 Métodos Especiales Comunes

- `__init__`: El constructor de la clase. Se llama cuando se crea un nuevo objeto de la clase.
- `__str__`: Define cómo se representa un objeto como una cadena de texto. Se llama cuando se usa la función `str()` o `print()` sobre el objeto.
- `__eq__`: Define cómo se comparan dos objetos para determinar si son iguales. Se llama cuando se usa el operador `==`.
- `__hash__`: Define el valor hash de un objeto. Es necesario para usar objetos como claves en diccionarios o en conjuntos.

1.4 Ejemplo en Python

A continuación, se muestra un ejemplo de cómo usar estos métodos especiales en el TDA `Punto`:

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

    def __eq__(self, otro_punto):
        if isinstance(otro_punto, Punto):
            return self.x == otro_punto.x and self.y == otro_punto.y
```

```
12          return False
13
14      def __hash__(self):
15          return hash((self.x, self.y))
```

En este ejemplo, hemos definido cómo se crea un `Punto`, cómo se representa como una cadena, cómo se compara con otro `Punto` y cómo se calcula su valor hash. Esto nos permite usar los objetos `Punto` de manera más natural y consistente.