16 Apéndice D: POO Java vs Python

Muchos programadores de Java en últimas fechas se trasladan a Python, a menudo luchan con el enfoque de Python para la programación orientada a objetos (POO). El enfoque para trabajar con objetos, tipos de variables y otras capacidades de lenguaje tomadas por Python vs Java es bastante diferente. Puede hacer que cambiar entre ambos lenguajes sea muy confuso.

Este apartado se compara y contrasta el soporte de programación orientada a objetos en Python vs Java. Al final, podremos aplicar nuestros conocimientos de programación orientada a objetos a Python al comprender cómo reinterpretar nuestra comprensión de los objetos de Java en Python y utilizar los objetos de una manera Pythonica.

A lo largo de este apartado, podremos:

- Construir una clase básica tanto en Java como en Python
- Explorar cómo funcionan los atributos de objeto en Python vs Java
- Comparar y contrastar los métodos de Java y las funciones de Python
- Descubrir los mecanismos de herencia y polimorfismo en ambos lenguajes
- Investigar la reflexión en Python vs Java
- Aplicar todo en una implementación de clase completa en ambos lenguajes

Este apartado no es una introducción a la programación orientada a objetos. Más bien, compara características y principios orientados a objetos de Python vs Java. Es recomendable que el lector tenga conocimiento de Java y también este familiarizado con la codificación de Python.

16.1 Ejemplo de Clases Java vs Python

Para comenzar, implementaremos la misma clase pequeña tanto en Java como en Python para ilustrar las diferencias entre ellos (se harán modificaciones a ellas a medida que avancemos).

Primero, supongamos que se tiene la siguiente definición de clase Carro en Java:

```
public class Carro {
   private String color;
   private String modelo;
   private int anio;
   public Car(String color, String modelo, int anio) {
      this.color = color;
      this.modelo = modelo;
      this.anio = anio;
   }
   public String getColor() {
      return color;
   }
   public String getmodelo() {
      return modelo;
   }
   public int getAnio() {
      return anio;
   }
}
```

Las clases de Java se definen en archivos con el mismo nombre que la clase. Entonces, se debe guardar esta clase en un archivo llamado *Carro.java*. Solo se puede definir una clase en cada archivo.

Una clase de Carro pequeña similar está escrita en Python de la siguiente manera:

```
class Carro:
   def __init__(self, color, modelo, anio):
      self.color = color
      self.modelo = modelo
      self.anio = anio
```

En Python se puede declarar una clase en cualquier lugar, en cualquier archivo, en cualquier momento. Por ejemplo, guardamos esta clase en el archivo carro.py.

Usando estas clases como base, podremos explorar los componentes básicos de clases y objetos.

16.2 Atributos de Objetos

Todos los lenguajes orientados a objetos tienen alguna forma de almacenar datos sobre el objeto. En Java y Python, los datos se almacenan en atributos, que son variables asociadas con objetos específicos.

Una de las diferencias más significativas entre Python y Java es cómo se definen y administran los atributos de clase y objeto. Algunas de estas diferencias provienen de las limitaciones impuestas por los lenguajes, mientras que otras provienen de las mejores prácticas de programación.

Declaración e Inicialización En Java, declaramos atributos en el cuerpo de la clase, fuera de cualquier método, con un tipo definido. Se debe definir los atributos de la clase antes de que se utilicen:

```
public class Carro {
   private String color;
   private String modelo;
   private int anio;
...
```

En Python, ambos se declaran y definen atributos dentro de la clase .__init__(), que es el equivalente al constructor de Java:

```
def __init__(self, color, modelo, anio):
   self.color = color
   self.modelo = modelo
   self.anio = anio
```

Al prefijar los nombres de las variables con *self*, le dice a Python que estos son atributos. Cada instancia de la clase tendrá una copia. Todas las variables en Python están escritas de forma flexible, y estos atributos no son una excepción.

También se puede crear variables de instancia fuera de .__init__(), pero no es una práctica recomendada ya que su alcance suele ser confuso. Si no se utilizan correctamente, las variables de instancia creadas fuera de .__init__() pueden provocar errores sutiles que son difíciles de encontrar. Por ejemplo, puede agregar un nuevo atributo .ruedas a un objeto *carro* como este:

```
1  >>> import carro
2  >>> mi_carro = carro.Carro("amarillo", "beetle", 1967)
3  >>> print(f"mi carro es {mi_carro.color}")
4  mi carro es amarillo
5
6  >>> mi_carro.ruedas = 5
7  >>> print(f"ruedas: {mi_carro.ruedas}")
8  ruedas: 5
```

Sin embargo, si olvidamos mi_carro.ruedas = 5 en la línea 6, Python muestra un error:

```
1  >>> import Carro
2  >>> mi_carro = Carro.Carro("amarillo", "beetle", 1967)
3  >>> print(f"Mi carro es {mi_carro.color}")
4  Mi carro es amarillo
5
6  >>> print(f"ruedas: {mi_carro.ruedas}")
7  Traceback (most recent call last):
8  File "<stdin>", line 1, in <module>
9  AttributeError: 'Carro' object has no attribute 'ruedas'
```

En Python, cuando declaras una variable fuera de un método, se trata como una variable de clase. Si actualizamos la clase Carro de la siguiente manera:

```
class Carro:
  ruedas = 0
  def __init__(self, color, modelo, anio):
    self.color = color
    self.modelo = modelo
    self.anio = anio
```

Esto cambia la forma en que usa la variable .ruedas. En lugar de referirse a él usando un objeto, se refiere a él usando el nombre de la clase:

```
1  >>> import carro
2  >>> mi_carro = carro.Carro("amarillo", "beetle", 1967)
3  >>> print(f"Mi carro es {mi_carro.color}")
4  Mi carro es amarillo
```

```
5
6  >>> print(f"Este tiene {carro.Carro.ruedas} ruedas")
7  Este tiene 0 ruedas
8
9  >>> print(f"Este tiene {mi_carro.ruedas} ruedas")
10  Este tiene 0 ruedas
```

Nota: En Python, se hace referencia a una variable de clase con la siguiente sintaxis:

El nombre del archivo que contiene la clase, sin la extensión .py
Un punto
el nombre de la clase
Un punto
El nombre de la variable

Como guardó la clase Carro en el archivo carro.py, se refiere a la variable de clase ruedas en la línea 6 como carro.Carro.ruedas.

Puedo consultar *mi_carro.ruedas* o *carro.Carro.ruedas*, pero hay que tener cuidado. Cambiar el valor de la variable de instancia *mi_carro.ruedas* no cambiará el valor de la variable de clase *carro.Carro.ruedas*:

```
1  >>> from carro import *
2  >>> mi_carro = carro.Carro("amarillo", "Beetle", "1966")
3  >>> mi_otro_carro = carro.Carro("rojo", "corvette",
"1999")
4
5  >>> print(f"Mi carro es {mi_carro.color}")
6  Mi carro es amarillo
7  >>> print(f"Este tiene {mi_carro.ruedas} ruedas")
8  Este tiene 0 ruedas
9
10  >>> print(f"Mi otro carro es {mi_otro_carro.color}")
11  Mi otro carro es rojo
12  >>> print(f"Este tiene {mi_otro_carro.ruedas} ruedas")
13  Este tiene 0 ruedas
```

```
14
15  >>> # cambiar el valor de la variable de clase
16  ... carro.Carro.ruedas = 4
17
18  >>> print(f"Este tiene {mi_carro.ruedas} ruedas")
19  Este tiene 4 ruedas
20  >>> print(f"Mi otro carro tiene {mi_otro_carro.ruedas}
ruedas")
21  Mi otro carro tiene 4 ruedas
22
23  >>> # Cambiar el valor de la variable de instancia para
mi_carro
24  ... mi_carro.ruedas = 5
25
26  >>> print(f"Mi carro tiene {mi_carro.ruedas} ruedas")
27  Mi carro tiene 5 ruedas
28  >>> print(f"Mi otro carro tiene {m_otro_carro.ruedas}
ruedas")
29  Mi otro carro tiene 4 ruedas
```

Se han definido dos objetos Coche en las líneas 2 y 3:

1. mi_carro
2. mi_otro_carro

Al principio, ambos tienen cero ruedas. Cuando establece la variable de clase usando *carro.Carro.ruedas = 4* en la línea 16, ambos objetos ahora tienen cuatro ruedas. Sin embargo, cuando configura la variable de instancia usando *mi_ carro.ruedas = 5* en la línea 24, solo ese objeto se ve afectado.

Esto significa que ahora hay dos copias diferentes del atributo de *ruedas*:

1. Una variable de clase que se aplica a todos los objetos *Carro*
2. Una variable de instancia específica aplicable solo al objeto *mi_ carro*

No es difícil referirse accidentalmente al incorrecto e introducir errores sutiles.

El equivalente de Java a un atributo de clase es un atributo estático:

```
public class Carro {
  private String color;
  private String modelo;
  private int anio;
  private static int ruedas;
  public Car(String color, String modelo, int anio) {
    this.color = color;
    this.modelo = modelo;
    this.anio = anio;
  }
  public static int getRuedas() {
    return ruedas;
  }
  public static void setRuedas(int count) {
    ruedas = count;
  }
}
```

Normalmente, se refiere a variables estáticas utilizando el nombre de clase de Java. Puede hacer referencia a una variable estática a través de una instancia de clase como Python, pero no es una práctica recomendada.

La clase de Java se está alargando. Una de las razones por las que Java es más detallado que Python es la noción de métodos y atributos públicos y privados.

Pública y Privada Java controla el acceso a métodos y atributos diferenciando entre datos públicos y datos privados.

En Java, se espera que los atributos se declaren como privados o protegidos si las subclases necesitan acceso directo a ellos. Esto limita el acceso a estos atributos desde el código fuera de la clase. Para proporcionar acceso a atributos privados, se declaran métodos públicos que establecen (*setters*) y recuperan (*getters)* datos de manera controlada.

Recordando de la clase Java anterior que la variable de color se declaró como privada. Por lo tanto, este código Java mostrará un error de compilación en la última línea:

```
Carro miCarro = new Carro("azul", "Ford", 1972);
// Pintando el carro
```

```
miCarro.color = "Rojo";
```

Si no especifica un nivel de acceso, el atributo predeterminado es paquete protegido, lo que limita el acceso a las clases en el mismo paquete. Debe marcar el atributo como público si desea que este código funcione.

Sin embargo, declarar atributos públicos no se considera una práctica recomendada en Java. Se espera que declare los atributos como privados y utilice métodos de acceso público, como *.getColor()* y *.getModelo()* que se muestran en el código.

Python no tiene la misma noción de datos privados o protegidos que tiene Java. Todo en Python es público. Este código funciona bien con su clase Python existente:

```
>>> mi_carro = Carro.Carro("azul", "Ford", 1972)
>>> # Pintando el carro
... mi_carro.color = "rojo"
```

En lugar de privada, Python tiene la noción de una variable de instancia no pública. Cualquier variable que comience con un carácter de subrayado se define como no pública. Esta convención de nomenclatura dificulta el acceso a una variable, pero es solo una convención de nomenclatura y aún puede acceder a la variable directamente.

Si Agregamos la siguiente línea a la clase Python Carro:

```
class Carro:
  ruedas = 0
  def __init__(self, color, modelo, anio):
    self.color = color
    self.modelo = modelo
    self.anio = anio
    self._portavasos = 6
```

Puede acceder a la variable ._portavasos directamente:

```
1  >>> import carro
2  >>> mi_carro = carro.Carro("amarillo", "Beetle", "1969")
3  >>> print(f"Fue construido en {mi_carro.anio}")
4  Fue construido en 1969
5  >>> mi_carro.anio = 1966
```

```
>>> print(f"Fue construido en {mi_carro.anio}")
Fue construido en 1966
>>> print(f"Tiene {mi_carro._portavasos} portavasos.")
Tiene 6 portavasos.
```

Python reconoce además el uso de caracteres de subrayado doble delante de una variable para ocultar un atributo en Python. Cuando Python ve una variable de subrayado doble, cambia el nombre de la variable internamente para dificultar el acceso directo. Este mecanismo evita accidentes pero aún así no imposibilita el acceso a los datos.

Para mostrar este mecanismo en acción, volvemos a cambiar la clase Python Carro:

```
class Carro:
  ruedas = 0
  def __init__(self, color, modelo, anio):
    self.color = color
    self.modelo = modelo
    self.anio = anio
    self.__portavasos = 6
```

Ahora, cuando se intenta acceder a la variable .__portavasos se ve el siguiente error:

```
>>> import Carro
>>> mi_carro = Carro.Carro("amarillo", "Beetle", "1969")
>>> print(f"Fue contruido en {mi_carro.anio}")
Fue construido en 1969
>>> mi_carro.anio = 1966
>>> print(f"Fue construido en {my_car.anio}")
Fue construido en 1966
>>> print(f"Tiene {mi_carro.__portavasos} portavasos.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carro' object has no attribute '__portava-
sos'
```

Entonces, ¿por qué no existe el atributo de .__portavasos?

Cuando Python ve un atributo con guiones bajos dobles, cambia el atributo anteponiendo el nombre original del atributo con un guión bajo, seguido del nombre de la clase. Para usar el atributo directamente, también debe cambiar el nombre que usa:

```
>>> print(f"Tiene {mi_carro._Carro__portavasos} portavasos")
Tiene 6 portavasos
```

Cuando usa guiones bajos dobles para ocultar un atributo al usuario, Python cambia el nombre de una manera bien documentada. Esto significa que un desarrollador determinado todavía puede acceder al atributo directamente.

Entonces, si sus atributos de Java se declaran privados y sus atributos de Python están precedidos por guiones bajos dobles, ¿cómo proporciona y controla el acceso a los datos que almacenan?.

Control de acceso En Java, se accede a atributos privados usando *getters* y se establecen usando *setters* de manera controlada. Para permitir que los usuarios pinten sus autos, agregando el siguiente código a la clase de Java:

```
public String getColor() {
  return color;
}
public void setColor(String color) {
  this.color = color;
}
```

Dado que *.getColor* () y *.setColor* () son públicos, cualquiera puede llamarlos para cambiar o recuperar el color del automóvil. Las mejores prácticas de Java de usar atributos privados a los que se accede con captadores *get* y definidores *set* públicos es una de las razones por las que el código Java tiende a ser más detallado que Python.

Como se vio anteriormente, acceder a los atributos directamente en Python. Dado que todo es público, se puede acceder a cualquier cosa en cualquier momento y desde cualquier lugar. Establece y obtiene valores de atributo directamente haciendo referencia a sus nombres. Incluso puede eliminar atributos en Python, lo que no es posible en Java:

```
>>> mi_carro = Carro("amarillo", "beetle", 1969)
>>> print(f"Mi carro fue construido en {mi_carro.anio}")
Mi carro fue construido en 1969
>>> mi_carro.anio = 1966
>>> print(f"Mi carro fue construido en {mi_carro.anio}")
Mi carro fue construido en 1966
>>> del mi_carro.anio
>>> print(f"Mi carro fue construido en {mi_carro.anio}")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carro' object has no attribute 'anio'
```

Sin embargo, es posible que en ocasiones desee controlar el acceso a un atributo. En ese caso, puede usar las propiedades de Python.

En Python, las propiedades proporcionan acceso controlable a los atributos de la clase mediante la sintaxis del decorador de Python. Las propiedades permiten que se declaren funciones en clases de Python que son análogas a los métodos getter y setter de Java, con la ventaja adicional de permitirle eliminar atributos también.

Podemos ver cómo funcionan las propiedades agregando una a la clase de Carro:

```
class Carro:
  def __init__(self, color, modelo, anio):
    self.color = color
    self.modelo = modelo
    self.anio = anio
    self._voltaje = 12
  @property
    def voltaje(self):
    return self._voltaje
  @voltaje.setter
  def voltaje(self, volts):
    print("Aviso: esto puede causar problemas!")
    self._voltaje = volts
  @voltaje.deleter
  def voltaje(self):
    print("Aviso: La radio dejara de funcionar!")
    del self._voltaje
```

Aquí, expandimos la noción de *Carro* para incluir vehículos eléctricos. Al Declarar el atributo *._voltaje* para mantener el voltaje de la batería.

Para proporcionar acceso controlado, definimos una función llamada *voltaje()* para devolver el valor privado. Al usar la decoración *@property*, lo marca como un captador al que cualquiera puede acceder directamente.

De manera similar, se define una función de establecimiento, también llamada *voltaje()*. Sin embargo, decoramos esta función con *@voltaje.setter*. Por último, usamos *@voltaje.deleter* para decorar un tercer *voltaje()*, lo que permite la eliminación controlada del atributo.

Los nombres de las funciones decoradas son todos iguales, lo que indica que controlan el acceso al mismo atributo. Los nombres de las funciones también se convierten en el nombre del atributo que se usa para acceder al valor. Así es como funcionan estas propiedades en la práctica:

```
1  >>> from carro import *
2  >>> mi_carro = Carro("amarillo", "beetle", 1969)
3
4  >>> print(f"Mi carro usa {mi_carro.voltaje} volts")
5  Mi carro usa 12 volts
6
7  >>> mi_carro.voltaje = 6
8  Aviso: esto puede causar problemas!
9
10  >>> print(f"Mi carro ahora usa {mi_carro.voltaje} volts")
11  Mi carro ahora usa 6 volts
12
13  >>> del mi_carro.voltaje
14  Aviso: la radio dejara de trabajar!
```

Tengamos en cuenta que usamos *.voltaje*, no *._voltaje*. Esto le dice a Python que use las funciones de propiedad que se definió:

- Cuando imprime el valor de mi_carro.voltaje en la línea 4, Python llama a *.voltaje()* decorado con *@property*.
- Cuando asigna un valor a mi_carro.voltaje en la línea 7, Python llama a *.voltaje()* decorado con *@voltaje.setter*.

- Cuando elimina *mi_ carro.voltaje* en la línea 13, Python llama a *.voltaje()* decorado con *@voltaje.deleter*.

Las decoraciones *@property*, *@.setter* y *@.delete*r permiten controlar el acceso a los atributos sin requerir que los usuarios utilicen métodos diferentes. Incluso puede hacer que los atributos parezcan propiedades de solo lectura omitiendo las funciones decoradas *@.setter* y *@.deleter*.

self and this En Java, una clase se refiere a sí misma con la referencia *this*:

```
public void setColor(String color) {
  this.color = color;
}
```

this está implícito en el código Java: normalmente no es necesario escribirlo, a menos que pueda haber confusión entre dos variables con el mismo nombre.

Podemos escribir el mismo código de esta manera:

```
public void setColor(String newColor) {
  color = newColor;
}
```

Dado que *Carro* tiene un atributo llamado *.color*, y no hay otra variable en el alcance con el mismo nombre, una referencia a ese nombre funciona. Usamos esto en el primer ejemplo para diferenciar entre el atributo y el parámetro, ambos con nombre color.

En Python, la palabra clave *self* tiene un propósito similar. Así es como se refiere a las variables de miembro, pero a diferencia de *this* de Java, es necesario si desea crear o hacer referencia a un atributo de miembro:

```
class Carro:
  def __init__(self, color, modelo, anio):
    self.color = color
    self.modelo = modelo
    self.anio = anio
    self._voltaje = 12
  @property
  def voltaje(self):
    return self._voltaje
```

Python requiere cada uno de ellos en el código anterior. Cada uno crea o hace referencia a los atributos. Si los omite, Python creará una variable local en lugar de un atributo.

La diferencia entre cómo se usa *self* y *this* en Python y Java se debe a diferencias subyacentes entre los dos lenguajes y cómo nombran variables y atributos.

16.3 Métodos y Funciones

Esta diferencia entre Python y Java es, en pocas palabras, que Python tiene funciones y métodos, mientras que Java sólo métodos.

En Python, el siguiente código está perfectamente bien (y es muy común):

```
>>> def di_hola():
... print("Hola!")
...
>>> di_hola()
Hola!
```

Se puede llamar a *di_hola()* desde cualquier lugar que sea visible. Esta función no tiene referencia a *self*, lo que indica que es una función global, no una función de clase. No puede alterar ni almacenar ningún dato en ninguna clase, pero puede usar variables locales y globales.

Por el contrario, cada línea de código Java que escribe pertenece a una clase. Las funciones no pueden existir fuera de una clase y, por definición, todas las funciones de Java son métodos. En Java, lo más cerca que puede llegar a una función pura es mediante el uso de un método estático:

```
public class Utils {
   static void DiHola() {
      System.out.println("Hola!");
   }
}
```

Utils.DiHola() se puede llamar desde cualquier lugar sin crear primero una instancia de *Utils*. Como puede llamar a *DiHola()* sin crear un objeto primero, esta referencia no existe. Sin embargo, esta todavía no es una función en el sentido de *di_hola()* de Python.

16.4 Herencia y Polimorfismo

La herencia y el polimorfismo son dos conceptos fundamentales en la programación orientada a objetos.

La herencia permite a los objetos derivar atributos y funcionalidad de otros objetos, creando una jerarquía que se mueve de objetos más generales a más específicos. Por ejemplo, un automóvil y un bote son tipos específicos de vehículos. Los objetos pueden heredar su comportamiento de un único objeto principal o de varios objetos principales, y se denominan objetos secundarios cuando lo hacen.

El polimorfismo permite que dos o más objetos se comporten entre sí, lo que permite que se usen indistintamente. Por ejemplo, si un método o función sabe cómo pintar un objeto Vehículo, también puede pintar un objeto Coche o Barco, ya que heredan sus datos y comportamiento del Vehículo.

Estos conceptos fundamentales de POO se implementan de manera bastante diferente en Python vs Java.

Herencia Python admite la herencia múltiple o la creación de clases que heredan el comportamiento de más de una clase principal.

Para ver cómo funciona esto, actualicemos la clase de Carro dividiéndola en dos categorías, una para vehículos y otra para dispositivos que usan electricidad:

```
class Vehiculo:
  def __init__(self, color, modelo):
    self.color = color
    self.modelo = modelo
class Dispositivo:
  def __init__(self):
    self._voltaje = 12
class Carro(Vehiculo, Dispositivo):
  def __init__(self, color, modelo, anio):
    Vehiculo.__init__(self, color, modelo)
    Dispositivo.__init__(self)
    self.anio = anio
  @property
  def voltaje(self):
    return self._voltaje
```

```
@voltaje.setter
def voltaje(self, volts):
  print("Aviso: esto puede causar problemas!")
  self._voltaje = volts
@voltaje.deleter
def voltaje(self):
  print("Aviso: el radio dejara de trabajar!")
  del self._voltaje
```

Un vehículo se define por tener atributos *.color* y *.modelo*. Entonces, un dispositivo se define para tener un atributo *._voltaje*. Dado que el objeto Carro original tenía estos tres atributos, se puede redefinir para heredar las clases Vehiculo y Dispositivo. Los atributos de color, modelo y _voltaje serán parte de la nueva clase Carro.

En *.__init__()* para Carro, llama a los métodos *.__init__()* para ambas clases principales para asegurarse de que todo se inicialice correctamente. Una vez hecho esto, puede agregar cualquier otra funcionalidad que desee a su Carro. En este caso, agrega un atributo *.anio* específico para los objetos Carro y los métodos *getter* y *setter* para *.voltaje*.

Funcionalmente, la nueva clase Carro se comporta como siempre. Creas y usa los objetos Carro como antes:

```
>>> from carro import *
>>> mi_carro = Carro("amarillo", "beetle", 1969)
>>> print(f"Mi carro es {mi_carro.color}")
Mi carro es amarillo
>>> print(f"Mi carro usa {mi_carro.voltaje} volts")
Mi carro usa 12 volts
>>> mi_carro.voltaje = 6
Aviso: esto puede causar problemas!
>>> print(f"Mi carro ahora usa {mi_carro.voltaje} volts")
Mi carro ahora usa 6 volts
```

Java, por otro lado, solo admite herencia única, lo que significa que las clases en Java pueden heredar datos y comportamiento de una sola clase principal. Sin embargo, los objetos Java pueden heredar el comportamiento de muchas interfaces diferentes. Las interfaces proporcionan un grupo de métodos relacionados que un objeto debe implementar y permiten que varias clases secundarias se comporten de manera similar.

Para ver esto en acción, divida la clase Java Carro en una clase principal y una interfaz:

```
public class Vehiculo {
  private String color;
  private String modelo;
  public Vehiculo(String color, String modelo) {
    this.color = color;
    this.modelo = modelo;
  }
  public String getColor() {
    return color;
  }
  public String getModelo() {
    return modelo;
  }
}
public interface Dispositivo {
  int getVoltaje();
}
public class Carro extends Vehiculo implements Dispositivo {
  private int voltaje;
  private int anio;
  public Car(String color, String modelo, int anio) {
    super(color, modelo);
    this.anio = anio;
    this.voltaje = 12;
  }
  @Override
  public int getVoltaje() {
    return voltaje;
  }
  public int getAnio() {
    return anio;
  }
}
```

Recordemos que cada clase e interfaz debe vivir en su propio archivo.

Como hicimos con Python, creamos una nueva clase llamada Vehículo para contener los datos y la funcionalidades más generales relacionados con el vehículo. Sin embargo, para agregar la funcionalidad del dispositivo, debe crear una interfaz en su lugar. Esta interfaz define un método único para devolver el voltaje del dispositivo.

La redefinición de la clase *Carro* requiere que heredes de *Vehiculo* usando extender e implemente la interfaz de *Dispositivo* usando implements. En el constructor, llama al constructor de la clase padre usando el *super()* incorporado. Dado que solo hay una clase principal, solo puede hacer referencia al constructor del vehículo. Para implementar la interfaz, escribe *getVoltaje()* usando la anotación *@Override*.

En lugar de obtener la reutilización del código de Dispositivo como lo hizo Python, Java requiere que implemente la misma funcionalidad en cada clase que implementa la interfaz. Las interfaces solo definen los métodos, no pueden definir datos de instancia o detalles de implementación.

Entonces, ¿por qué es este el caso de Java? Todo se reduce a tipos.

Tipos y Polimorfismo La estricta verificación de tipos de Java es lo que impulsa el diseño de su interfaz.

Cada clase e interfaz en Java es un tipo. Por lo tanto, si dos objetos Java implementan la misma interfaz, se considera que son del mismo tipo con respecto a esa interfaz. Este mecanismo permite usar indistintamente diferentes clases, que es la definición de polimorfismo.

Puede implementar la carga del dispositivo para sus objetos Java creando un .cargar() que necesita un dispositivo para cargarse. Cualquier objeto que implemente la interfaz del dispositivo se puede pasar a .cargar(). Esto también significa que las clases que no implementan Dispositivo generarán un error de compilación.

Cree la siguiente clase en un archivo llamado Rinoceronte.java:

```
public class Rinoceronte {
}
```

Ahora puede crear un nuevo *Main.java* para implementar *.cargar()* y explorar en qué se diferencian los objetos Carro y Rinoceronte:

```
public class Main{
```

```
    public static void cargar(Dispositivo dispositivo) {
      dispositivo.getVoltaje();
    }
    public static void main(String[] args) throws Exception {
      Carro carro = new Carro("amarillo", "beetle", 1969);
      Rinoceronte rinoceronte = new Rinoceronte();
      cargar(carro);
      cargar(rinoceronte);
    }
}
```

Esto es lo que debería ver cuando intente compilar este código:

```
Information:2021-02-02 15:20 - Compilation completed with
1 error and 0 warnings in 4 s 395 ms
Main.java
Error:(43, 11) java: incompatible types: Rinoceronte cannot
be converted to Dispositivo
```

Dado que la clase Rinoceronte no implementa la interfaz del dispositivo, no se puede pasar a *.cargar()*.

En contraste con la escritura de variables estricta de Java, Python usa un concepto llamado escritura de pato, que en términos básicos significa que si una variable "camina como un pato y grazna como un pato, entonces es un pato". En lugar de identificar objetos por tipo, Python examina su comportamiento.

Podemos explorar la escritura de pato implementando capacidades de carga de dispositivo similares para la clase de Dispositivo Python:

```
>>> def cargar(dispositivo):
... if hasattr(Dispositivo, '_voltaje'):
... print(f"Cargando a {dispositivo._voltaje} voltaje disposi-
tivo ")
... else:
... print(f"No se puede cargar {dispositivo.__class__.__name__}")
...
>>> class Telefono(Dispositivo):
... pass
```

```
...
>>> class Rinoceronte:
... pass
...
>>> mi_carro = Carro("amarillo", "Beetle", "1966")
>>> mi_telefono = Telefono()
>>> mi_Rinoceronte = Rinoceronte()
>>> cargar(mi_carro)
Cargando 12 volts dispositivo
>>> cargar(mi_telefono)
Cargando a 12 volts Dispositivo
>>> cargar(mi_Rinoceronte)
No se puede cargar Rinoceronte
```

cargar() debe verificar la existencia del atributo *._voltaje* en el objeto al que se le pasa. Dado que la clase Dispositivo define este atributo, cualquier clase que herede de él (como Carro y Telefono) tendrá este atributo y, por lo tanto, mostrará que se está cargando correctamente. Las clases que no heredan de Dispositivo (como Rinoceronte) pueden no tener este atributo y no podrán cargar (lo cual es bueno, ya que cargar rinocerontes puede ser peligroso).

Métodos Predeterminados Todas las clases de Java descienden de la clase Object, que contiene un conjunto de métodos que heredan todas las demás clases. Las subclases pueden anularlas o mantener los valores predeterminados. La clase Object define los siguientes métodos:

```
class Object {
  boolean equals(Object obj) { ... }
  int hashCode() { ... }
  String toString() { ... }
}
```

De forma predeterminada, *equals()* compara las direcciones del objeto actual con un segundo objeto pasado, y *hashcode()* calcula un identificador único que también usa la dirección del objeto actual. Estos métodos se utilizan en muchos contextos diferentes en Java. Por ejemplo, las clases de

servicios públicos, como las colecciones que clasifican objetos en función del valor, necesitan ambas.

toString() devuelve una representación de cadena del objeto. De forma predeterminada, este es el nombre de la clase y la dirección. Este método se llama automáticamente cuando se pasa un objeto a un método que requiere un argumento de cadena, como *System.out.println()*:

```
Carro carro = new Carro("amarillo", "Beetle", 1969);
System.out.println(carro);
```

Al ejecutar este código, se usará el *.toString()* predeterminado para mostrar el objeto de Carro:

```
Carro@61bbe9ba
```

No es muy útil, ¿verdad? Puede mejorar esto anulando el *.toString()* predeterminado. Agregue este método a la clase Java Carro:

```
public String toString() {
   return "Carro: " + getColor() + " : " + getmodelo() + "
: " + getanio();
}
```

Ahora, cuando ejecutemos el mismo código de muestra, veremos lo siguiente:

```
Carro: amarillo : Beetle : 1969
```

Python proporciona una funcionalidad similar con un conjunto común de *dunder* (abreviatura de "doble subrayado") de métodos. Cada clase de Python hereda estos métodos y puede anularlos para modificar su comportamiento.

Para las representaciones de cadena de un objeto, Python proporciona __repr__() y *__str__()*, que puede conocer en Pythonic OOP String Conversion: *__repr__* vs *__str__*. La representación inequívoca de un objeto es devuelta por *__repr__()*, mientras que *__str__()* devuelve una representación legible por humanos. Estos son aproximadamente análogos a *.hashcode()* y *.toString()* en Java.

Al igual que Java, Python proporciona implementaciones predeterminadas de estos métodos *dunder*:

```
>>> mi_carro = Carro("amarillo", "Beetle", "1966")
>>> print(repr(mi_carro))
<Carro.Carro object at 0x7fe4ca154f98>
>>> print(str(mi_carro))
<Carro.Carro object at 0x7fe4ca154f98>
```

Puede mejorar esta salida anulando *.__str__()*, agregando esto a la clase Python Carro:

```
def __str__(self):
  return f'Carro {self.color} : {self.modelo} : {self.anio}'
```

Esto le da un resultado mucho mejor:

```
>>> mi_carro = Carro("amarillo", "Beetle", "1966")
>>> print(repr(mi_carro))
<Carro.Carro object at 0x7f09e9a7b630>
>>> print(str(mi_carro))
Carro amarillo : Beetle : 1966
```

Anular el método *dunder* nos dio una representación más legible de su Carro. Es posible que también desee anular el *.__repr__()*, ya que a menudo es útil para depurar.

Python ofrece muchos más métodos *dunder*. Usando métodos *dunder*, puede definir el comportamiento de su objeto durante la iteración, comparación, adición o hacer que un objeto sea invocable directamente, entre otras cosas.

Sobrecarga del Operador La sobrecarga de operadores se refiere a redefinir cómo funcionan los operadores de Python cuando operan en objetos definidos por el usuario. Los métodos *dunder* de Python le permiten implementar la sobrecarga de operadores, algo que Java no ofrece en absoluto.

Modificando la clase Python Carro con los siguientes métodos adicionales:

```
class Carro:
  def __init__(self, color, modelo, anio):
    self.color = color
    self.modelo = modelo
    self.anio = anio
```

```
def __str__(self):
  return f'Carro {self.color} : {self.modelo} : {self.anio}'
def __eq__(self, other):
  return self.anio == other.anio
def __lt__(self, other):
  return self.anio < other.anio
  def __add__(self, other):
  return Carro(self.color + other.color,
    self.modelo + other.modelo,
    int(self.anio) + int(other.anio))
```

La siguiente tabla muestra la relación entre estos métodos *dunder* y los operadores de Python que representan:

Método Dunder	Operator	Propósito
__eq__	==	Revisa si son del mismo año
__lt__	<	Cual carro es un modelo más reciente
__add__	+	Concatena dos objetos Carro de una manera absurda

Cuando Python ve una expresión que contiene objetos, llama a los métodos *dunder* definidos que correspondan a los operadores de la expresión. El siguiente código utiliza estos nuevos operadores aritméticos sobrecargados en un par de objetos Carro:

```
>>> mi_carro = Carro("amarillo", "Beetle", "1966")
>>> tu_carro = Carro("rojo", "Corvette", "1967")
>>> print (mi_carro < tu_carro)
True
>>> print (mi_carro > tu_corro)
False
>>> print (mi_carro == tu_carro)
False
>>> print (mi_carro + tu_Carro)
Car amarillorojo : BeetleCorvette : 3933
```

Hay muchos más operadores que puede sobrecargar usando métodos *dunder*. Ofrecen una forma de enriquecer el comportamiento de su objeto de una manera que los métodos predeterminados de la clase base común de Java no lo hacen.

16.5 Reflexión

La reflexión se refiere a examinar un objeto o clase desde dentro del objeto o clase. Tanto Java como Python ofrecen formas de explorar y examinar los atributos y métodos de una clase.

Examinar el Tipo de un Objeto Ambos lenguajes tienen formas de probar o verificar el tipo de un objeto.

En Python, usa *type()* para mostrar el tipo de una variable, e *isinstance()* para determinar si una variable dada es una instancia o hija de una clase específica:

```
>>> mi_carro = Carro("amarillo", "Beetle", "1966")
>>> print(type(mi_carro))
<class 'Carro.Carro'>
>>> print(isinstance(mi_carro, Carro))
True
>>> print(isinstance(mi_carro, Dispositivo))
True
```

En Java, consulta el objeto por su tipo usando *.getClass()*, y usa el operador *instanceof* para verificar una clase específica:

```
Carro carro = new Carro("amarillo", "beetle", 1969);
System.out.println(carro.getClass());
System.out.println(carro instanceof Carro);
```

Este código genera la siguiente salida:

```
class com.realpython.Carro
true
```

Examinar los Atributos de un Objeto En Python, puede ver todos los atributos y funciones contenidos en cualquier objeto (incluidos todos los métodos *dunder*) usando *dir()*. Para obtener los detalles específicos de un atributo o función determinados, use *getattr()*:

```
>>> print(dir(mi_carro))
['_Carro__portavasos', '__add__', '__class__', '__de-
lattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__',
'__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__re-
duce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__',
'_voltaje', 'color', 'modelo', 'voltaje', 'ruedas', 'anio']
>>> print(getattr(mi_carro, "__format__"))
<built-in method __format__ of Carro object at 0x7fb4c10f5438>
```

Java tiene capacidades similares, pero el control de acceso del lenguaje y la seguridad de tipos lo hacen más complicado de recuperar.

.getFields() recupera una lista de todos los atributos de acceso público. Sin embargo, dado que ninguno de los atributos de Carr es público, este código devuelve una matriz vacía:

```
Field[] fields = carro.getClass().getFields();
```

Java trata los atributos y métodos como entidades independientes, por lo que los métodos públicos se recuperan mediante *.getDeclaredMethods()*. Dado que los atributos públicos tendrán un método *.get* correspondiente, una forma de descubrir si una clase contiene una propiedad específica podría verse así:

> Utilice *.getDeclaredMethods()* para generar una matriz de todos los métodos.
>
> Recorra todos los métodos devueltos:
>
> Para cada método descubierto, devuelve verdadero si el método:
>
> Comienza con la palabra get OR acepta cero argumentos
>
> Y no vuelve vacío
>
> Y incluye el nombre de la propiedad
>
> De lo contrario, devuelve falso.

Aquí hay un ejemplo rápido y sucio:

```
    public static boolean getProperty(String name, Object ob-
ject) throws Exception {
      Method[] declaredMethods = object.getClass().getDeclaredMethods();
      for (Method method : declaredMethods) {
        if (isGetter(method) &&
          method.getName().toUpperCase().contains(name.toUpperCase()))
{
            return true;
        }
      }
      return false;
    }
    // Función auxiliar para obtener si el método es un método
getter
    public static boolean isGetter(Method method) {
      if ((method.getName().startsWith("get") ||
        method.getParameterCount() == 0 ) &&
        !method.getReturnType().equals(void.class)) {
          return true;
      }
      return false;
    }
```

getProperty() es su punto de entrada. Llamandolo con el nombre de un atributo y un objeto. Devuelve verdadero si se encuentra la propiedad y falso si no.

Métodos de Llamada a Través de la Reflexión Tanto Java como Python proporcionan mecanismos para llamar a métodos mediante la reflexión.

En el ejemplo de Java anterior, en lugar de simplemente devolver verdadero si se encuentra la propiedad, puede llamar al método directamente. Recuerde que *getDeclaredMethods()* devuelve una matriz de objetos *Method*. El objeto *Method* en sí tiene un método llamado *.invoke()*, que llamará al método. En lugar de devolver true cuando el método correcto se encuentra en la línea 7 anterior, puede devolver *method.invoke(object)* en su lugar.

Esta capacidad también existe en Python. Sin embargo, dado que Python no distingue entre funciones y atributos, debe buscar específicamente entradas que sean invocables:

```
>>> for method_name in dir(mi_carro):
... if callable(getattr(mi_carro, method_name)):
... print(method_name)
...
__add__
__class__
__delattr__
__dir__
__eq__
__format__
__ge__
__getattribute__
__gt__
__init__
__init_subclass__
__le__
__lt__
__ne__
__new__
__reduce__
__reduce_ex__
__repr__
__setattr__
__sizeof__
__str__
__subclasshook__
```

Los métodos de Python son más simples de administrar y llamar que en Java. Agregar el operador () (y cualquier argumento requerido) es todo lo que necesita hacer.

El siguiente código encontrará el *.__str__ ()* de un objeto y lo llamará a través de la reflexión:

```
>>> for method_name in dir(mi_carro):
... attr = getattr(mi_carro, method_name)
... if callable(attr):
... if method_name == '__str__':
... print(attr())
...
Carro amarillo : Beetle : 1966
```

Aquí, se comprueba cada atributo devuelto por *dir()*. Obtiene el objeto de atributo real usando *getattr()* y verifica si es una función invocable usando *callable()*. Si es así, compruebe si su nombre es *__str__()* y luego llámelo.