ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 8

Імплементація ООП (продовження)

Весна 2021

PyToken Class

Об'єктно-орієнтоване програмування полягає у створенні об'єктів. Об'єкти мають інформацію про стан, яку іноді називають просто станом, і методи, які діють на цей стан, іноді змінюючи стан. Якщо ми змінюємо стан об’єкта, ми називаємо його змінним об’єктом. Якщо ми не можемо змінити стан об’єкта після його створення, об’єкт називається незмінним.

Клас визначає інформацію про стан, що підтримується об’єктом, і методи, що діють на цей стан. Ми почнемо з вивчення класу **`PyToken`** на рис. 8.1, оскільки це простий незмінний клас. Клас **`PyToken`** визначає маркери-об'єкти, які є простим незмінним класом. Клас **`PyToken`** визначає об'єкти маркерів, які сканер повертає синтаксичному аналізатору під час синтаксичного аналізу програми JCoCo. Весь код JCoCo міститься в пакеті під назвою **`jcoco`**. Рядок 1 заявляє, що цей клас належить до цього пакета **`jcoco`**.

Рис.8.1
PyToken class

```java
package jcoco;
public class PyToken {

    public enum TokenType {
        PYIDENTIFIERTOKEN,
        PYINTEGERTOKEN,
        PYFLOATTOKEN,
        PYSTRINGTOKEN,
        PYKEYWORDTOKEN,
        PYCOLONTOKEN,
        PYCOMMATOKEN,
        PYSLASHTOKEN,
        PYLEFTPARENTOKEN,
        PYRIGHTPARENTOKEN,
        PYEOFTOKEN,
        PYBADTOKEN
    }

    private String lexeme;
    private TokenType type;
    private int line;
    private int col;

    public PyToken(TokenType type, String lex, int line, int col) {
        this.lexeme = lex;
        this.type = type;
        this.line = line;
        this.col = col;
    }

    public TokenType getType() {
        return this.type;
    }

    public String getLex() {
        return this.lexeme;
    }

   // See getCol and getLine in the full source code.
}
```

Рядки 4–17 на рис. 8.1 визначають перелік **`TokenType`**, що є коротким для перерахування. Перелік **`TokenType`** визначає типи маркерів, що повертаються сканером. Якщо згадати з попередніх лекцій, синтаксис програм CASM досить простий, і ці постійні значення є всіма можливими типами маркерів. Кожна константа служить іменем для кожного типу лексеми. За допомогою цього переліку можна використовувати ці імена констант у коді Java там, де потрібен тип маркера. Наприклад, ось один фрагмент коду з інтерпретатора JCoCo. Використання описових назв констант корисно при написанні коду самодокументування, до якого вам слід завжди прагнути.

```
if (tok.getType() != TokenType.PYEOFTOKEN)
{
    badToken(tok, "Excpected End Of File (EOF)");
}
```

Рядки 19–22 визначають змінні екземпляра об’єкта (тобто стан об’єкта). Кожна з цих змінних оголошується приватною, так що лише методи класу можуть безпосередньо отримувати доступ до інформації про стан. Рядки 24–29 визначають конструктор **PyToken**, який викликається при створенні об’єкта **PyToken**. Ось як створюється об’єкт **PyToken**.

```
PyToken t;
t = new PyToken(type, lex, line, column);
```

Звичайно, змінні тип, **lex**, рядок і стовпець вже мали б мати значення і бути відповідними типами. Наприклад, змінна **lex** повинна бути оголошена як **String**. Java - це статично набрана мова, тому всі змінні повинні бути оголошені перед тим, як їх можна використовувати. У цьому коді було оголошено, що змінна t має тип **PyToken**.

Є кілька речей, яких ви не можете побачити в підручнику. Оскільки цей клас визначений у пакеті під назвою **`jcoco`**, результат компіляції цього класу буде розміщений у підкаталозі з назвою **`jcoco`**. Пакети та підкаталоги поєднуються в Java-організацію файлів. Крім того, цей файл повинен мати ім'я **`PyToken.java`**, оскільки він містить загальнодоступний клас **`PyToken`**. Це також є частиною організації файлів Java.

C++ PyToken Class

Реалізація **`PyToken`** в C ++ виглядає дещо інакше, ніж версія Java. Java і C ++ підтримують окрему компіляцію коду. За допомогою Java кожен клас записується в окремий файл. Кожен файл компілюється окремо компілятором Java. Коли повторно компілювати клас Java, вирішується на основі дат як файлів **`.java`**, так і **`.class`**.

У C ++ немає такого механізму **make**, вбудованого в компілятор. Натомість окремий інструмент make забезпечує цю функціональність, як було описано в лекції 7. Крім того, інтерфейс до класу (тобто оголошення змінних та методів екземпляра класу) відокремлений від фактичного коду, який реалізує методи. Отже, визначення класу **PyToken** розділено на два файли: файл заголовка **PyToken**, який називається **PyToken.h**, та реалізації методу, розташовані в **PyToken.cpp**.

На рисунку 8.2 показано вміст оголошення файлу заголовка класу. За допомогою компілятора компілюються лише вихідні файли `.cpp`. Заголовкові файли включаються у вихідні файли для використання під час компіляції вихідних файлів.

Рис.8.2 Опис C++ PyToken class — PyToken.h

```
#include <string>
using namespace std;
class PyToken {
public:
   PyToken(int tokenType, string lex, int line, int col);
   virtual ~PyToken();
   string getLex() const;
   int getType() const;
   int getCol() const;
   int getLine() const;
private:
   string lexeme;
   int tokenType;
   int line;
   int column;
};
const int PYIDENTIFIERTOKEN = 1;
const int PYINTEGERTOKEN = 2;
const int PYFLOATTOKEN = 3;
const int PYSTRINGTOKEN = 4;
const int PYKEYWORDTOKEN = 5;
const int PYCOLONTOKEN = 6;
const int PYCOMMATOKEN = 7;
const int PYSLASHTOKEN = 8;
const int PYLEFTPARENTOKEN = 9;
const int PYRIGHTPARENTOKEN = 10;
const int PYEOFTOKEN = 98;
const int PYBADTOKEN = 99;
```

Визначення класу **PyToken** розділено на два файли: файл заголовка **PyToken**, що називається **PyToken.h**, та реалізації методів, розташовані в **PyToken.cpp**. На рисунку 8.2 показано вміст оголошення файлу заголовка класу. За допомогою компілятора компілюються лише вихідні файли **.cpp**. Заголовкові файли включаються у вихідні файли для використання під час компіляції вихідних файлів.

Значна частина оголошення класу у файлі заголовка виглядає як версія Java. Перелік, визначений на рис. 8.1, реалізований як цілі константи на рис. 8.2 без поважних причин. Перерахування також підтримуються на C++. У рядку 6 подано декларацію деструктора, який, як правило, використовується C++, оскільки програми C ++ повинні звільнити свій простір, оскільки немає збирача сміття, як у Java. Однак у цьому випадку деструктор насправді не має мети, оскільки ці маркери не мають вказівників на інші об’єкти. Деструктор потрібен саме тоді, коли об'єкт містить вказівники на інші ресурси, які потрібно звільнити.

Об'єкти **PyToken** не містять жодних покажчиків на інші об'єкти, а отже, деструктор не має призначення в цьому класі.
Інша різниця, на яку варто звернути увагу, - це використання **const** після чотирьох методів, що повертають значення. Це заявляє, що ці методи не мутують об'єкт **PyToken**.
Вони повертають значення лише з об'єкта **PyToken**. Використання **const** існує в C++, оскільки C++ дуже гнучкий у способі передачі параметрів та повернення значень. Оголошення, що метод є **const**, допомагає C++ знати, де метод можна безпечно викликати, і може оптимізувати роботу програм на C++.

```cpp
#include "PyToken.h"
PyToken::PyToken(int tokenType, string lex, int line, int col) {
    this->lexeme = lex;
    this->tokenType = tokenType;
    this->line = line;
    this->column = col;
}
PyToken::~PyToken() {}
int PyToken::getType() const {
    return tokenType;
}
string PyToken::getLex() const {
    return lexeme;
}
// getLine and getCol omitted.
```

Рис. 8.3 Імплементація C++ PyToken class — PyToken.cpp

Реалізація **PyToken** на C++ наведена на рис. 8.4. Перший рядок містить декларацію файлу заголовка **PyToken.h**. Це директива макропроцесора, яка додає вихідний код до цього файлу. Роблячи це, для цього вихідного коду оголошено клас **PyToken**.

PyToken::, який ви бачите на рис. 8.4, є кваліфікатором сфери. Це вказує на те, що, хоча жоден з цього коду не записаний фізично всередині визначення класу **PyToken**, він призначений як частина класу **PyToken**.

У рядках 3–6 на рис. 8.3 використовується оператор стрілки, написаний `->`. На Java це написано крапкою. Оператор стрілки слідує за вказівником. У C++ це вказівник, який вказує на поточний об'єкт. У Java це посилання, і ми використовуємо крапкові позначення для розмежування посилання на це посилання. Покажчики - це адреса даних у пам’яті комп’ютера. Покажчики можна використовувати у виразах для створення нових покажчиків за допомогою арифметики покажчиків. У мові програмування вказівник може вказувати куди завгодно. Посилання набагато більш контрольоване. Посилання дещо нагадують покажчики, за винятком того, що їх не можна використовувати в арифметичних виразах.

Вони також не вказують безпосередньо на місця в пам'яті. Коли посилання розрізнюються за допомогою крапки, система часу виконує пошук у таблиці посилань.
Ця різниця між посиланнями та покажчиками означає, що ми можемо спокійно покладатися на кожне посилання, що вказує на реальний об'єкт, де ми не обов'язково знаємо, чи вказує вказівник на простір, який може бути безпечно звільнений чи ні, оскільки покажчик може бути результатом деяких арифметика покажчика. Посилання безпечні для збору сміття. Покажчики — ні.

Наслідування та поліморфізм

Об'єктно-орієнтовані мови програмування допомагають нам упорядкувати наш код. Одна велика перевага організації нашого коду навколо об’єктів виникає, коли ми можемо повторно використовувати код. Повторне використання коду важливо, щоб ми могли щось написати один раз і забути про це під час вирішення інших проблем. Але для повторного використання коду нам потрібен спосіб налаштування цього коду для наших цілей.

Спадкування - це механізм, який ми використовуємо для повторного використання коду в програмному забезпеченні, яке ми зараз пишемо. *Поліморфізм* - це механізм, який ми використовуємо для налаштування поведінки коду, який ми вже писали. У цій лекції ми розглянемо деякі коди на C ++, щоб побачити, як визначаються успадкування та поліморфізм. У наступному розділі ми знову переглянемо той самий код, що реалізований на Java.

Розглянемо файл заголовка для **PyObject** на рис. 8.4. Віртуальні машини CoCo та JCoCo працюють над об’єктами Python. Кожне значення даних у Python є об'єктом, тому ця ідея об'єктів є дуже поширеною в реалізаціях JCoCo / CoCo. Насправді це настільки поширене, що є певні речі, які повинен робити кожен об’єкт у Python. Для кожного об’єкта Python можна викликати певні методи. Щоб мати змогу повторно використовувати якомога більше коду, має сенс написати цей загальний код в одному місці. Одне з таких місць - клас **PyObject** у реалізації C++ CoCo.

Рис.8.4

```
#ifndef PYOBJECT_H_
#define PYOBJECT_H_

#include <string>
#include <unordered_map>
#include <vector>
#include <iostream>
using namespace std;

class PyType;

class PyObject {
public:
   PyObject();
   virtual ~PyObject();
   virtual PyType* getType();
   virtual string toString();
   PyObject* callMethod(string name, vector<PyObject*>* args);

protected:
    unordered_map<string, PyObject* (PyObject::*)(vector<PyObject*>*)> dict;
    virtual PyObject* __str__(vector<PyObject*>* args);
    virtual PyObject* __type__(vector<PyObject*>* args);
    virtual PyObject* __hash__(vector<PyObject*>* args);
    virtual PyObject* __repr__(vector<PyObject*>* args);
};

ostream& operator << (ostream& os, PyObject& t);
extern bool verbose;
#endif /* PYOBJECT_H_ */
```

Кожен об'єкт у Python може бути перетворений у рядок. Хоча подання рядків різняться, механізм перетворення об'єкта в рядок полягає у виклику методу **__str__** на об'єкті. Це заявлено в рядку 22 на рис. 8.4. Оскільки всі об'єкти повинні реагувати на цей метод, клас **PyObject** визначає метод **toString** та **__str__**, який викликає метод **toString**. Метод **__repr__** подібний до методу **__str__**. У деяких випадках два методи повертають абсолютно однаковий рядок. Але **__repr__** повертає рядок, який, якщо обчислюється за допомогою функції **eval**, створить той самий об'єкт. Наприклад, розглянемо цей код.

```
x = [1,2,3]
y = eval(repr(x))
```

Після оцінки цього коду як **x**, так і **y** посилаються на списки цілих чисел, де **y** - повна копія вмісту списку, на який посилається **x**.

Кожен об'єкт у Python відповідає на ряд основних викликів методів. CoCo не намагається реалізувати всі з них. Але іншим, що він реалізує, є метод **__hash__**, який повертає хеш-значення для всіх хеш-об'єктів у Python.

Це використовується, коли об’єкт використовується як ключ у словнику (тобто хеш-таблиця). Тільки незмінні об’єкти можуть використовуватися як ключі у словниках. Метод `__type__` повертає тип будь-якого об'єкта Python. Кожен об'єкт Python має тип. Тип повертається за допомогою цього методу, який також є об'єктом.

На рис. 8.4 є кілька речей, що стосуються програмування на C++. Перші сім рядків називаються директивами макропроцесора. Будь-який рядок, що починається зі знака фунта (тобто **#**), є директивою макропроцесора. Перший рядок - це директива **if-not-defined**, а другий рядок - директива **define**. Останній рядок на рис. 8.4 - це закінчення, яке відповідає першому рядку.

Шаблон директив макропроцесора **ifndef, define, endif** необхідний, оскільки директиви **include** часто закінчуються круговими посиланнями, де включають **A** включає **B**, що, в свою чергу, може включати **C**, що включає **A** знову. Цього кругового посилання можна уникнути, визначивши **PYOBJECT _H _** на рис. 8.4.

Після включення **PyObject.h include** визначається **PYOBJECT _H _**, і якщо **PyObject.h** знову включається через кругову посилання або навіть через інше включення, включаючи його без кругової посилання, він не буде включений двічі. Цей шаблон **ifndef-define-endif** використовується для кожного файлу заголовка в програмуванні на C та C ++.

У рядку 10 оголошено клас (тобто тип), який називається **`PyType`**. Це називається прямою декларацією. Клас **`PyType`** використовується в цьому заголовковому файлі, але **`PyType.h`** також включає **`PyObject.h`**, тому декларація вперед була необхідною через циклічне посилання. Рядок 14 є конструктором для класу **`PyObject`**. Рядок 15 - це деструктор, який знову нічого не робить для цього класу.

Важливим є використання ключового слова **`virtual`**. Віртуальні методи - це методи, які включені до таблиці віртуальних функцій класу C++. Ця віртуальна таблиця функцій - це те, як C++ реалізує поліморфізм. Коли викликається віртуальна функція, відбувається додатковий пошук адреси функції, оскільки класи, які успадковуються від цього класу, можуть замінити будь-яку віртуальну функцію. Наприклад, під час компіляції не може бути відомо, яку версію **`toString`** слід викликати, версію **`PyObject`** або один із методів **`toString`**, визначені в підкласі **`PyObject`**.

Вивчіть метод **PyObject:: _ _str _ _**, показаний на рис. 8.5. Цей метод викликає **toString** і повертає новий об'єкт **PyStr** в результаті перетворення об'єкта в рядок. Тут тонко є те, що **toString** буде викликано, невідомо, поки цей код насправді не виконається. Наприклад, якщо поточним об'єктом є **PyInt**, тоді код буде виконуватися з **PyInt.cpp**, як показано на рис. 8.6. Але якби **PyList** був поточним об’єктом, тоді метод **toString** був би виконаний з рис. 8.7.

```
1  PyObject* PyObject::__str__(vector<PyObject*>* args) {
2      ostringstream msg;
3
4      if (args->size() != 0) {
5          msg << "TypeError: expected 0 arguments, got " << args->size();
6          throw new PyException(PYWRONGARGCOUNTEXCEPTION,msg.str());
7      }
8
9      return new PyStr(toString());
10 }
```

Рис. 8.5 CoCo str magic method

```
string PyInt::toString() {
    stringstream ss;
    ss << val;
    return ss.str();
}
```

Рис.8.6 PyInt toString method

```
string PyList::toString() {
    ostringstream s;
    vector<PyObject*> args;
    s << "[";
    for (int i=0;i<data.size();i++) {
        s << *(data[i]->callMethod("__repr__",&args));
        if (i < data.size()-1)
            s << ", ";
    }
    s << "]";
    return s.str();
}
```

Рис.8.7 PyList toString method

І **`PyInt`**, і **`PyList`** успадковують від **`PyObject`** у реалізації C++. Отже, поліморфізм працює, оскільки **`toString`** оголошено віртуальним, і тому визначення того, який **`toString`** викликати, здійснюється за допомогою додаткового пошуку фактичного вказівника на функцію у віртуальній таблиці функцій під час виконання.

Рядок 28 оголошує функцію, яка в даному випадку є перевантаженим оператором зсуву вліво (тобто **<<)**, який можна використовувати для друку об'єктів. Реалізація цього перевантаженого оператора зсуву вліво, з файлу **PyObject.cpp**, покладається на поліморфізм для налаштування своєї поведінки, як метод **__str__**, також реалізований у цьому модулі. **ToString**, який потрібно викликати, буде залежати від того, який тип об'єкта це викликається.

```
ostream& operator <<(ostream &os, PyObject &t) {
    return os << t.toString();
}
```

Інтерфейси і адаптери

У ранніх об'єктно-орієнтованих мовах програмування специфікація інтерфейсу була прив'язана безпосередньо до класу. Наприклад, у попередньому розділі ми дізналися, що **`toString`** було прив'язане безпосередньо до класу **`PyObject`** та будь-яких класів, які успадкували від **`PyObject`**. Це чудово працює, поки у вас не буде класу, який успадковує щось інше, ніж **`PyObject`**, і хотів би використовувати поліморфізм, визначений методом **`toString`**.

Тоді у вас виникає ситуація, коли ви хотіли б успадкувати від двох різних класів одночасно. C ++ вирішує цю проблему за допомогою багаторазового успадкування. Класи C ++ можуть успадковуватись від декількох класів.

Java робить кілька речей, дещо відрізняються від C ++. По-перше, на відміну від C ++, кожен клас у Java успадковує від класу **`Object`** прямо чи опосередковано. У Java існує одна ієрархія класів, у якій бере участь кожен клас. C ++ не має вбудованої ієрархії успадкування. Використовуючи C ++, клас, який явно не успадковує щось, не успадковує нічого. У Java клас, який явно не успадковує нічого, що успадковується від **`Object`**.

По-друге, багаторазове успадкування не підтримується за допомогою Java, що спрощує успадкування та його реалізацію. Але Java вирішує всю проблему інтерфейсів, прив'язаних до оголошень класів, розділяючи ці два поняття. Інтерфейс - це обіцянка підтримувати певні методи в класі. Класи можуть реалізовувати скільки завгодно інтерфейсів, що є способом Java для досягнення багаторазового успадкування. Але інтерфейси жодним чином не прив’язані до ієрархії класів. Більше того, ви можете оголосити параметр типом інтерфейсу. Розглянемо код на рис. 8.8.

Як і версія C ++, інтерфейс Java **`PyObject`** оголошує **`str ethod`**, схожий за призначенням на метод C ++ **`toString`**. Оголошення методу **`str`** в інтерфейсі означає, що всі класи, які реалізують цей інтерфейс, повинні реалізовувати метод **`str`**.

Хоча оголошення інтерфейсу відокремлює інтерфейс від ієрархії класів, воно також не реалізує жодного коду для інтерфейсу. Часто буває, що багато класів, які реалізують інтерфейс, матимуть принаймні якийсь загальний код. Кожен клас повинен реалізовувати один і той же код, або програміст може вибрати використання спадщини для написання класу адаптера, який реалізує інтерфейс і надає загальний код для декількох підкласів. Це так на рис. 8.9. Значна частина коду тут пропущена для стислості.

```
1  package jcoco;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5
6  public interface PyObject {
7      public String str() ;
8      public PyType getType();
9
10     public void set(String key, PyObject value) ;
11     public PyObject get(String key) ;
12     public PyObject callMethod(String name, ArrayList<PyObject> args) ;
13 }
```

Рис.8.8 Наслідування PyObject

```java
package jcoco;

public class PyObjectAdapter implements PyObject {
    protected HashMap<String, PyObject> dict = new HashMap<String, PyObject>();
    protected HashMap<String, PyObject> attrs = new HashMap<String, PyObject>();
    protected String name;
    protected PyType.PyTypeId type;

    public PyObjectAdapter(String name, PyType.PyTypeId type) {
        this();
        this.name = name;
        this.type = type;
    }

    public PyObjectAdapter() {
        name = "PyObject()";
        type = PyType.PyTypeId.PyClassType;
        PyObjectAdapter self = this;
        this.dict.put("__str__", new PyBaseCallable() {
            @Override
            public PyObject __call__(ArrayList<PyObject> args) {
                if (args.size() != 0) {
                 throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                        "TypeError: expected 0 argument, got " + args.size());
                }

                return new PyStr(self.str());
```

Рис. 8.9

```
            }
        });
        ...
    }
```

PyObjectAdapter

```
    @Override
    public PyType getType() {
        return JCoCo.PyTypes.get(type);
    }

    @Override
    public String str() {
        return name;
    }

    @Override
    public String toString() {
        PyStr s = (PyStr)callMethod("__repr__",new ArrayList<PyObject>());
         return s.str();
    }

    @Override
    public void set(String key, PyObject value) {
        this.dict.put(key, value);
    }
    ...
}
```

У цьому коді є кілька речей, на які слід звернути увагу. Рядки 4–7 визначають захищені змінні. Захищені змінні приховані (тобто недоступні) від будь-якого коду, який не знаходиться в одному пакеті, в даному випадку пакету `jcoco`. Рядок 10 коду викликає `this ()`, який викликає конструктор за замовчуванням у рядку 15, тому код, загальний для обох конструкторів, не потрібно дублювати.

У рядку 33 використовується декоратор під назвою `@Override`. Цей декоратор повідомляє компілятору, що метод, який слідує, замінює або реалізує метод з базового класу або інтерфейсу. Це корисно, якщо ви допускаєте орфографічну помилку або неправильно вказуєте тип параметра методу. Неправильне написання імені або зміна типу параметра призведе до методу, який ніколи не буде викликаний, оскільки поліморфізм працює лише тоді, коли ім’я та типи аргументів збігаються.

В іншому випадку ви просто визначаєте інший метод, оскільки Java підтримує параметричне перевантаження імен, тобто два методи можуть мати одне і те ж ім’я, якщо вони мають різні типи параметрів. Отже, `@Override` може бути корисним для виявлення помилок, які в іншому випадку можуть бути важкими для налагодження.

Функції як величини

Python - це мова, що динамічно набирається. Як такі, методи шукаються під час виконання, а не під час компіляції. Усі методи та значення в Python - це об'єкти, які зберігаються у словниках в інших об'єктах, щоб їх можна було переглянути під час виконання. Ключі в цих словниках - це рядки: імена значень, методів або функцій. Для реалізації віртуальної машини, яка працює як віртуальна машина Python, необхідно розглядати функції як значення і зберігати їх у словниках або хеш-таблицях, як реалізація віртуальної машини Python. C ++ підтримує розгляд функцій як значень. Використовуючи C ++, ми можемо написати наступне.

```
dict["__str__"]=(PyObject* (PyObject::*)(vector<
PyObject*>*)) (&PyObject::__str__);
```

Цей код походить із класу PyObject у реалізації CoCo на C ++, у файлі **PyObject.cpp**. У цьому коді є тонкий нюанс. Після встановлення методу **_ _str _ _** у словнику об'єкта будь-які та всі підкласи, які успадковуються від **PyObject** і замінюють метод **__str__**, автоматично отримують визначення перевизначеного методу. Немає необхідності встановлювати "**__str__**", щоб вказувати на новий, перевизначений **__str__** метод.

Оскільки метод `__str__` оголошений віртуальним, поліморфізм означає, що його потрібно встановити у словнику лише один раз.

Код **`SImilar`** у Java неможливий, оскільки Java не розглядає функції та методи як значення. Але рішення є. Клас може імітувати функцію або метод.

Насправді Java містить підтримку саме цього. JCoCo реалізує власну версію під час виконання методу або функції під час виконання, використовуючи об'єкти для імітації функцій і методів.

```
1 public interface PyCallable extends PyObject {
2     public PyObject __call__(ArrayList<PyObject> args) ;
3 }
```

Рис.8.10 Інтерфейс PyCallable

JCoCo визначає інтерфейс під назвою **PyCallable**, показаний на рис. 8.10. Цей інтерфейс визначає один із методів **__call__**. Цей метод бере список посилань на **PyObject** і повертає **PyObject** як його результат. Це відображає механізм виклику в Python.

Усі функції Python отримують список об’єктів Python і повертають об’єкт Python. Ця однорідність означає, що будь-який клас, який реалізує інтерфейс **`PyCallable`**, можна викликати, викликаючи його метод **`_ _call _ _`**. Це означає, що функції можна розглядати як значення в JCoCo, кодуючи функції як об'єкти.

Анонімні внутрішні класи

Інтерфейси в Java визначають методи, які повинні підтримуватися класами, що їх реалізують. Інтерфейс **`PyCallable`**, описаний в останньому розділі, визначає метод **`__call__`**. Як і інші інтерфейси, існують класи адаптерів, які надають певний загальний код для класів, які вирішили реалізувати інтерфейс **`PyCallable`**. Найпростіший з них - клас **`PyBaseCallable`**, який використовується у функціях, реалізованих у класі **`PyObjectAdapter`** та класі **`PyCallableAdapter`**, щоб уникнути проблеми кругового посилання в ієрархії об'єктів.

Інший клас також реалізує інтерфейс **`PyCallable`** під назвою **`PyCallableAdapter`**, який використовується всіма іншими реалізаціями **`PyCallable`**, крім тих, що створені в класах **`PyObjectAdapter`** та **`PyCallableAdapter`**.

Рисунок 8.9 містить код у рядках 19–29, який створює екземпляр адаптера **PyBase-Callable**. Це приклад анонімного внутрішнього класу. Рядок 19 створює клас, який не має імені, але успадковується від **PyBaseCallable** і замінює метод **__call__**. Існує один екземпляр цього створеного класу **PyBaseCallable**, екземпляр для цього методу **__str__**. Коли об'єкт бажає викликати метод **__str__** на об'єкті, він викликає метод **callMethod** класу **PyObjectAdapter**.

```
@Override
public PyObject callMethod(String name, ArrayList<PyObject> args) {
    PyCallable mbr = null;
    if (this.dict.containsKey(name)) {
        mbr = (PyCallable) this.dict.get(name);
        return mbr.__call__(args);
    }
    throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
        "TypeError: '" + this.getType().str() +
        "' object has no attribute '" + name + "'");
}
```

Рис. 8.11

Анонімні внутрішні класи дуже важливі для Java. Внутрішній клас - це будь-який клас, визначений у межах іншого класу. Внутрішні класи важливі, оскільки вони забезпечують засоби реалізації зворотних дзвінків. Коли подія відбувається в програмі Java, як подія із програми графічного інтерфейсу (тобто клацання мишею) або повідомлення, що надходить з Інтернету, якщо для обробки цієї події було зареєстровано зворотний дзвінок, то зворотний дзвінок є зателефонувавшим. Внутрішні класи є ідеальним способом реалізації зворотного виклику, оскільки внутрішній клас автоматично має доступ до всіх змінних та методів зовнішнього об'єкта.

У випадку з кодом на рис. 8.9 внутрішній клас є анонімним, він не має імені. Це нормально, оскільки зазвичай для зворотного виклику створено один і лише один екземпляр для конкретного зовнішнього об’єкта. У попередніх версіях Java не було анонімних класів, що залишало програми Java, заповнені внутрішніми класами, які коли-небудь створювали лише один екземпляр. Програмістам Java потрібен був більш компактний, точний синтаксис для управління зворотними дзвінками та були введені анонімні класи.

Перевага внутрішнього класу, визначеного в рядках 19–29 на рис. 8.9, полягає в рядку 27, де викликається метод **`str ()`**, який є членом **`PyObjectAdapter`**, що є класом зовнішнього об'єкта в даному випадку. Оскільки це внутрішній клас, ми можемо безпосередньо викликати метод **`str ()`** у цьому зворотному дзвінку. Анонімні внутрішні класи широко використовуються впродовж реалізації JcoCo.

Приведення типів і дженериків

Рядок 21 коду C ++ на рис. 8.8 є приклад оголошення змінної **`dict`**, тип якої визначається шаблоном. Шаблон - це те, як програмісти на C ++ пишуть загальні класи. Загальний клас - це зазвичай контейнер якогось типу, в даному випадку хеш-таблиця. Ця хеш-таблиця відображає рядки у функції, яким надається вектор вказівників **`PyObject`**, і повертає вказівник **`PyObject`**. Клас вектора - знову шаблон. Вектор, переданий цим функціям, є послідовністю покажчиків **`PyObject`**.

Дженерики є важливою частиною об'єктно-орієнтованих мов. Дженерики дозволяють програмістам повторно використовувати класи, особливо класи, які розроблені як структури даних, такі як карти та вектори. Карта - це структура даних, що відображає ключі до значень. Тип ключів і значень може бути практично будь-яким. Отже, загальний клас карти забезпечує можливість зіставити будь-який тип ключів із будь-яким типом значень. На Java карта називається **`HashMap`**. У C ++ існує кілька видів класів карт. Зверніть увагу, що в C ++ стандартний клас map не реалізований як хеш-таблиця. Це гарантує час вставки та пошуку **`O (log n)`**.

Хеш-таблиця гарантує амортизовану складність вставки та пошуку `O (1)`. Невпорядкована _map C ++ 11 реалізована як хеш-таблиця. До C ++ 11 цей клас не входив до C ++.

Java має один тип ієрархії. Все успадковує від **`Object`** прямо чи опосередковано. Маючи ієрархію одного типу, творці Java можуть забезпечити класи контейнерів для багатьох структур даних, які нам потрібні в наших програмах, включаючи **`HashMap`** та **`ArrayList`**, що забезпечує засіб для зберігання списку об’єктів. Наприклад, якщо вам потрібен список об'єктів, які потрібно повернути з функції, ви можете кодувати його, як показано на рис. 8.12.

```
1  private ArrayList BodyPart() {
2          ArrayList instructions = new ArrayList();
3          PyToken tok = this.in.getToken();
4          this.target.clear();
5          this.index = 0;
6          if (!tok.getLex().equals("BEGIN")) {
7              badToken(tok, "Expected a BEGIN keyword.");
8          }
9          ....
```

Рис. 8.12. Приклад ArrayList

Функція **BodyPart**, яка є частиною модуля **PyParser**, повертає список об'єктів **PyByteCode**. Коли потрібен один із цих об'єктів **PyByteCode**, ми будемо змушені написати такий код, щоб отримати доступ до першого об'єкту **PyByteCode** у списку.

```
ArrayList bp = BodyPart();
PyByteCode byteCode = (PyByteCode) bp.get(0);
```

(**`PyByteCode`**) з паренами називається типовим приводом або просто приводом. Кастинг необхідний при переміщенні вниз по ієрархії успадкування. Привід - це спосіб сказати компілятору Java, що ви знаєте фактичний тип значення, тоді як компілятор цього не знає. Існує перевірка часу виконання, яка вставляється у ваш код. Якщо пристрій недійсний, програма Java видасть виняток, тому кастинг безпечний. Це просто не зручно, і додаткова перевірка часу роботи є менш бажаною, хоча, можливо, краще, ніж не перевірка взагалі.

Кастинг однаковий у C ++ та Java. Та сама проблема виникає в C ++. При переміщенні вниз по ієрархії успадкування буде потрібний привід. C ++ має тип даних, подібний до Java **`ArrayList`**, який називається вектор. Однак C ++ не має одного суперкласу з усіх інших класів. Тож векторний тип даних було б трохи важче написати без чогось, що називається дженериками.

Переміщення ієрархії успадкування в Java або C ++ вимагає від програміста написання додаткового коду. Якби нам спочатку вдалося уникнути переміщення вгору по ієрархії, тоді рух знову вниз став би непотрібним. Це мета дженериків. Загальні засоби були додані до Java, щоб зробити переміщення вгору і вниз по ієрархії успадкування, а отже, і лиття, непотрібним у багатьох обставинах. Розглянемо реальну версію функції **`BodyPart`** на рис. 8.13.

```
1       private ArrayList<PyByteCode> BodyPart() {
2           ArrayList<PyByteCode> instructions = new ArrayList<PyByteCode>();
3           PyToken tok = this.in.getToken();
4           this.target.clear();
5           this.index = 0;
6           if (!tok.getLex().equals("BEGIN")) {
7               badToken(tok, "Expected a BEGIN keyword.");
8           }
9           ...
```

Рис. 8.13 Приклад ArrayList, що використовує дженерики.

```
template < class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator< pair<const Key,T> >
           > class unordered_map {
           ...
}
```

Рис. 8.14 Шаблон unordered _map

У цьому коді кутові дужки (тобто **<i>**) обмежують тип **ArrayList**. **ArrayList** - це список елементів **PyByteCode**. Це оголошує конкретний тип, що міститься в **ArrayList**, роблячи декларацію загального **ArrayList**, так що він може бути контейнером будь-якого типу, а не лише значенням **Object**. Щоб оголосити клас **ArrayList** загальним, творці Java змінили б його визначення таким чином.

```
class ArrayList<T> {
    private T data[] = new T[10];
    ...
```

Тип **T** стає параметром для оголошення класу. Для кожної заявленої версії **ArrayList** створюється версія класу. Отже, коли вказано клас **ArrayList <PyByteCode>**, створюється об’єкт **ArrayList PyByteCode**.

Python, оскільки він динамічно набирається, не потребує загальних засобів. Загальні засоби потрібні лише для статично набраних мов, таких як Java та C ++. У C ++ дженерики називаються шаблонами. Шаблон - це параметризований клас. Як і Java, параметром класу є тип або типи. Стандартні контейнери шаблонів у C ++ включають невпорядковану **_map**, карту, вектор, список, чергу, стек, deque, set та масив серед інших.

Розглянемо декларацію невпорядкованого шаблону **_map** у C++ на рис. 8.14. Це визначення шаблону показує нам, що більше ніж один параметр типу можна використовувати для шаблону або загального в C++ або Java. У випадку невпорядкованого **_map** існує п'ять параметрів типу, що передаються до декларації карти.

Алмазна нотація - одна з тем, пов’язана із дженериками в Java. Щоб ще більше заощадити написання, програмісти Java можуть використовувати алмазні позначення під час написання загальної декларації об’єкта. Декларація вказівок змінної раніше в цьому розділі могла б бути написана наступним чином, якби ми використовували алмазні позначення.

```
ArrayList<PyByteCode> instructions = new ArrayList<>();
```

Ви, напевно, бачите брилік у коді. Оскільки **`PyByteCode`** вже написано один раз у цьому рядку, за допомогою Java вам не доведеться писати його знову. Компілятор може зробити висновок про тип **`ArrayList`**, оскільки він створюється на основі типу посилальних інструкцій, що вказують на нього.

На наступній лекції продовжимо розгляд імплементації об’єкто-орієнтованого програмування.