Java程式設計基礎 - Kibo-RPC教學

1：變數與資料儲存

- **標題：** 變數 - 程式中的資料儲存容器
- **內容：**
  - 變數是程式中用來儲存資料的「容器」
  - 每個變數都有三個基本特性：
    1. 名稱（識別符）：如何稱呼這個變數，例如 `age`、`name`、`position`
    2. 資料型態：決定變數可以存放什麼類型的資料
    3. 值：變數實際儲存的資料
  - 變數的值可以在程式執行過程中改變（這也是為什麼稱為「變」數）
  - 宣告變數的基本語法：

```
資料型態 變數名稱;  // 單純宣告
資料型態 變數名稱 = 初始值;  // 宣告並賦值
```

  - 宣告變數的例子：

```
int age = 25;                   // 整數變數設值為25
String name = "Astrobee";       // 字串變數設值為"Astrobee"
double distance = 10.5;         // 雙精度浮點數設值為10.5
```

  - 變數使用原則：
    - 變數名稱應該有意義，反映其用途
    - Java的變數名稱區分大小寫（`age` 和 `Age` 是不同的變數）
    - 變數在使用前必須先宣告並賦值
  - 賦值運算子(=)：將右側的值賦給左側的變數

```
speed = 10;          // 將speed的值改為10
speed = speed + 2;   // 將speed的值加2
speed += 5;          // 等同於 speed = speed + 5
```

  - 常見錯誤：在條件判斷中誤用`=`（賦值）而非`==`（比較）

2：Java識別符命名規則

- **標題：** Java識別符命名規則與風格
- **內容：**
  - **開頭字元 (Starts With):**
    - 必須是： 字母 (A-Z, a-z)、金錢符號 ($) 或底線 (_)。

    - 強烈建議： 以 字母 開頭。
  - **後續字元 (Subsequent Chars):**
    - 可以是：字母、數字 (0-9)、金錢符號 ($) 或底線 (_)。
  - **禁止使用 (Cannot Be):**
    - Java 關鍵字 (e.g., int, class, public, static, if, for).
    - 空格或 $ 和 _ 以外的特殊符號。
  - **大小寫敏感 (Case-Sensitive):**
    - myVariable 與 MyVariable 是 不同 的。
  - **長度 (Length):**
    - 無明確限制，但以簡潔且有意義為佳。
  - **識別符類型與慣例風格:**

識別符類型 (Identifier Type)	慣例風格 (Convention Style)	範例 (Examples)
變數 (Variables)	小駝峰式 (lowerCamelCase)	userName, age, targetPoint, totalScore
方法 (Methods)	小駝峰式 (lowerCamelCase)	calculateArea(), getUserName(), runPlan1()
類別 (Classes)	大駝峰式 (UpperCamelCase/PascalCase)	MainActivity, YourService, Student, Point
介面 (Interfaces)	大駝峰式 (UpperCamelCase/PascalCase)	Runnable, List, Comparable, ActionListener
常數 (Constants) (final)	全大寫 + 底線 (UPPER_SNAKE_CASE)	MAX_USERS, DEFAULT_PORT, PI
套件 (Packages)	全小寫 (alllowercase)	java.util, org.example.project, com.myapp

3：基本資料型態

- **標題：** 基本資料型態概覽
- **內容：**
  - **數值型態**
    - `int`：整數型態（沒有小數點的數字）

```
int count = 42;          // 正整數
int temperature = -7;    // 負整數
```

- 範圍：約±21億（-2,147,483,648 至 2,147,483,647）
- 用途：計數、索引、簡單計算

- `double`：雙精度浮點數（有小數點的數字，高精度）

```
double price = 19.99;           // 金額
double pi = 3.14159265359;      // 高精度數值
```

- 範圍：約±1.7×10^308，精度約15-16位數
- 用途：需要小數點的計算、科學計算

- `float`：單精度浮點數（有小數點的數字，精度較低）

```
float temperature = 36.5f;  // 注意後面的'f'是必須的
```

- 範圍：約±3.4×10^38，精度約6-7位數
- 用途：需要小數點但不需高精度的情況

- **布林型態**

- `boolean`：布林型態（只有兩種值：true或false）

```
boolean isMoving = true;        // 機器人正在移動
boolean missionComplete = false; // 任務尚未完成
```

- 用途：表示開/關、是/否、真/假等二元狀態
- 在條件判斷和迴圈控制中非常重要

- **文字型態**

- `String`：字串型態（文字序列）

```
String robotName = "Astrobee";  // 機器人名稱
String combined = robotName + " is ready!"; // 字串可以連接
```

- 用途：儲存和處理文字資訊
- 字串用雙引號 " 括起來

- `char`：字元型態（單一字符）

```
char grade = 'A';       // 英文字母
```

    - 用途：表示單個字符
    - 字元用單引號 ' 括起來

4：Kibo-RPC特有的資料型態

- **標題：** Kibo-RPC特有的資料型態
- **內容：**
    - `Point`：三維空間中的點（位置）

```
Point position = new Point(10.5, -9.8, 4.3);
// x=10.5, y=-9.8, z=4.3，表示太空站內的一個位置
```

        - 包含三個座標分量：x, y, z
        - 用於`moveTo()`方法來控制機器人移動
    - `Quaternion`：四元數（旋轉/方向）

```
Quaternion orientation = new Quaternion(0, 0, -0.707, 0.707);
// 表示機器人在太空中的方向
```

        - 包含四個分量：x, y, z, w
        - 比尤拉角(Euler angles)更適合表示三維空間中的旋轉
        - 在Kibo-RPC中用於控制機器人的朝向
    - `Mat`：圖像矩陣（相機捕獲的影像）

```
Mat image = api.getMatNavCam();
// 獲取導航攝像機的圖像
```

        - 來自OpenCV函式庫，用於圖像處理
        - 在Kibo-RPC中用於讀取和分析目標物體

5：複合資料結構

- **標題：** 複合資料結構：陣列與集合
- **內容：**
    - **陣列(Array)**：儲存相同型態的多個值的容器

```
int[] numbers = {1, 2, 3, 4, 5};  // 宣告並初始化整數陣列
String[] items = {"crystal", "emerald", "diamond"};  // 字串陣列
```

        - 特性：

    - 固定大小（一旦創建，長度不可改變）
    - 索引從0開始（第一個元素的索引是0）
    - 每個元素必須是相同型態
  - 存取與修改陣列元素：

```
int firstNumber = numbers[0];  // 獲取第一個元素
numbers[2] = 30;  // 修改元素
int arrayLength = numbers.length;  // 獲取陣列長度
```

  - 二維陣列：

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6}
};
int value = matrix[1][2];  // 獲取值為6
```

- **List集合**：有序的元素集合，與陣列類似但大小可變

```
List<String> itemNames = new ArrayList<>();  // 宣告字串List
List<Integer> scores = new ArrayList<>();    // 整數List(注意使用Integer
而非int)
```

  - 特性：
    - 大小可變（可以動態增加或減少元素）
    - 有序（保持元素的插入順序）
    - 只能存儲物件，不能直接存儲基本型態
  - 常用方法：

```
itemNames.add("crystal");           // 添加元素
String firstItem = itemNames.get(0); // 獲取元素
itemNames.remove("diamond");        // 移除元素
int size = itemNames.size();        // 獲取大小
```

- **Map集合**：用於存儲鍵值對(key-value pairs)

```
Map<String, Integer> itemCounts = new HashMap<>();  // 宣告Map

// 添加鍵值對
itemCounts.put("crystal", 2);
itemCounts.put("emerald", 1);
```

```
// 獲取值
int count = itemCounts.get("crystal");  // 獲取值為2
```

- 特性：
  - 每個鍵都是唯一的
  - 每個鍵映射到一個值
  - 無序（除非使用LinkedHashMap）

- **Set集合**：不允許重複元素的集合

```
Set<String> uniqueItems = new HashSet<>();

uniqueItems.add("crystal");
uniqueItems.add("emerald");
uniqueItems.add("crystal");  // 重複元素不會被添加
```

- 特性：
  - 不允許重複元素
  - 只關心元素是否存在，不關心位置

6：布林運算與流程控制

- **標題**： 布林運算與流程控制
- **內容**：

  - **布林運算子**：

    - &&：邏輯AND（兩條件都為true，結果才為true）

```
if (isFound && isMissionComplete) {
    // 只有當找到目標且任務完成時才執行
}
```

    - ||：邏輯OR（任一條件為true，結果就為true）

```
if (isEmergency || timeLimit < 10) {
    // 當緊急情況或時間少於10秒時執行
}
```

    - !：邏輯NOT（反轉布林值）

```
if (!isFound) {
    // 當尚未找到目標時執行
}
```

- **條件判斷：**

```
if (condition) {
    // 當條件為true時執行
} else if (anotherCondition) {
    // 當第一個條件為false，第二個條件為true時執行
} else {
    // 當以上條件都為false時執行
}
```

- **迴圈控制：**
  - while迴圈：當條件為true時重複執行

```
while (!isMissionComplete) {
    // 當任務未完成時，重複執行這段程式碼
}
```

  - for迴圈：常用於已知迭代次數的情況

```
for (int i = 0; i < 5; i++) {
    // 執行5次
}

// 遍歷集合或陣列
for (String item : itemNames) {
    // 處理每個item
}
```

7：String字串操作詳解

- **標題：** String字串操作詳解
- **內容：**
  - String是Java中表示文字的資料型態（是一個類別而非基本型態）
  - 字串是不可變的(immutable)—修改字串實際上是創建新字串
  - 字串的常用方法：

```
String message = "Hello, Astrobee!";

// 取得字串長度
int length = message.length();  // 返回15
```

```
// 取得特定位置的字元（索引從0開始）
char firstChar = message.charAt(0);  // 返回'H'

// 檢查字串是否包含某子字串
boolean contains = message.contains("Astro");  // 返回true

// 字串比較
boolean isEqual = message.equals("Hello, Astrobee!");  // 比較內容，返回
true

// 分割字串
String[] parts = message.split(", ");  // 返回["Hello", "Astrobee!"]

// 擷取子字串
String sub = message.substring(7, 15);  // 返回"Astrobee"

// 轉換大小寫
String lower = message.toLowerCase();  // 返回"hello, astrobee!"
String upper = message.toUpperCase();  // 返回"HELLO, ASTROBEE!"

// 移除首尾空白
String trimmed = "  text  ".trim();  // 返回"text"

// 替換字串
String replaced = message.replace("Hello", "Hi");  // 返回"Hi,
Astrobee!"
```

- 字串連接：

```
// 使用 + 運算子
String fullName = firstName + " " + lastName;

// 使用StringBuilder（更高效，適合多次連接）
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(", ");
sb.append("Astrobee");
String result = sb.toString();  // 返回"Hello, Astrobee"
```

- 字串與其他型態的轉換：

```
// 數字轉字串
String numStr = String.valueOf(42);  // 整數轉字串
String doubleStr = String.valueOf(3.14);  // 浮點數轉字串

// 字串轉數字
int num = Integer.parseInt("42");  // 字串轉整數
double dbl = Double.parseDouble("3.14");  // 字串轉浮點數
```

9：註解與程式碼風格

- **標題：** 註解與程式碼風格
- **內容：**
  - **註解**是給人閱讀的說明文字，電腦會忽略這些內容
  - 註解的作用：
    - 解釋程式碼的用途和邏輯
    - 暫時「關閉」不需要的程式碼
    - 幫助團隊成員理解你的思路
    - 提醒自己特定的實作細節
  - 單行註解：使用 `//`

```
// 這是一個說明下一行程式碼作用的註解
int count = 0;  // 這個註解位於程式碼行尾
```

  - 多行註解：使用 `/* ... */`

```
/* 這是一個可以跨越
   多行的註解，適合較長的
   說明和解釋 */
```

  - 文檔註解：使用 `/** ... */`（用於生成API文檔）

```
/**
 * 計算兩點間的距離
 * @param p1 第一個點
 * @param p2 第二個點
 * @return 兩點間的歐幾里得距離
 */
double calculateDistance(Point p1, Point p2) {
    // 方法實作...
}
```

  - **程式碼風格：**
    - 使用有意義的變數名稱
    - 保持一致的縮排（通常使用4個空格或一個tab）
    - 每個敘述後使用分號結尾
    - 大括號通常位於同一行或下一行
    - 適當使用空白行分隔邏輯區塊

10：方法的基本結構

- **標題：** 方法（函數）的基本結構
- **內容：**
  - 方法（Method）是執行特定任務的程式碼區塊
  - 方法的基本結構：

```
返回型態 方法名稱(參數型態 參數名稱, ...) {
    // 方法主體（程式碼區塊）
    // 執行特定任務的程式碼...

    return 返回值;  // 如果返回型態不是void，必須有return陳述句
}
```

  - 方法結構解析：
    - **返回型態**：方法完成後返回的資料型態（int、String等）或void（不返回任何值）
    - **方法名稱**：命名應具有描述性，通常使用小駝峰式（首單字小寫，後續單字首字母大寫）
    - **參數列表**：方法需要的輸入資料（可以沒有參數）
    - **方法主體**：實際執行的程式碼
    - **return陳述句**：指定返回給呼叫者的值（void方法可省略）
  - 方法宣告範例：

```
// 無參數，無返回值的方法
void startMission() {
    // 啟動任務的程式碼
}

// 有參數，有返回值的方法
double calculateDistance(Point p1, Point p2) {
    // 計算兩點間距離的程式碼
    double distance = Math.sqrt(Math.pow(p2.getX() - p1.getX(), 2) +
                                Math.pow(p2.getY() - p1.getY(), 2) +
                                Math.pow(p2.getZ() - p1.getZ(), 2));
    return distance;
}
```

11：方法的呼叫與參數傳遞

- **標題：** 方法的呼叫與參數傳遞
- **內容：**
  - 呼叫方法的基本語法：

```
方法名稱(參數1, 參數2, ...);  // 不關心返回值

返回型態 變數名 = 方法名稱(參數1, 參數2, ...);  // 儲存返回值
```

- 呼叫方法的範例：

```
// 呼叫無參數的方法
startMission();

// 呼叫有參數的方法
Point p1 = new Point(0, 0, 0);
Point p2 = new Point(3, 4, 0);
double dist = calculateDistance(p1, p2);  // 會返回5.0

// 使用方法返回值進行條件判斷
if (calculateDistance(p1, p2) < 6.0) {
    // 當距離小於6.0時執行的程式碼
}
```

- 方法參數傳遞規則：
  - 基本資料型態（int、double等）傳遞的是值的複製（傳值）
  - 物件型態（String、Point等）傳遞的是參考（物件的記憶體位址）
- Kibo-RPC中的方法呼叫：

```
// 呼叫API方法
api.startMission();
api.moveTo(new Point(11.0, -5.7, 4.5),
          new Quaternion(0, 0, -0.707, 0.707),
          false);

// 呼叫方法並存儲返回值
Mat image = api.getMatNavCam();
```

12：方法的細節與建議

- **標題**： 方法的細節與設計建議
- **內容**：
  - 方法命名建議：
    - 使用動詞或動詞短語
    - 具體描述方法的功能
    - 遵循Java命名規範（小駝峰式）
    - 例如：`calculateDistance`, `moveForward`, `processImage`
  - 參數設計建議：
    - 參數數量適中（通常不超過3-4個）
    - 參數順序合理、一致

        - 參數名稱清晰描述其用途

    - 方法設計原則：

        - 單一職責：一個方法只做一件事
        - 適當大小：通常一個方法不應太長（30-50行為宜）
        - 降低複雜度：過於複雜的邏輯應拆分為多個方法

    - 覆載（Overloading）方法：同名但參數不同的多個方法

```
// 計算兩點距離
double calculateDistance(Point p1, Point p2) {
    // ...計算程式碼
}

// 計算當前位置到指定點的距離
double calculateDistance(Point target) {
    Point current = getCurrentPosition();
    return calculateDistance(current, target);
}
```

    - 在Kibo-RPC中的應用：

        - 將複雜任務分解成多個方法便於管理
        - 提高程式碼可讀性和可維護性
        - 例如：可以分別實作`scanArea()`, `identifyTarget()`, `moveToTarget()`等方法

13：類別（Class）的概念

- **標題：** 類別（Class）的基本概念
- **內容：**

    - 什麼是類別？

        - 類別是物件的藍圖或模板
        - 定義了物件所具有的資料（屬性）和行為（方法）
        - Java是物件導向程式語言，幾乎所有東西都是物件

    - 類別的基本結構：

```
public class Robot {
    // 屬性（成員變數）：描述物件的特性
    private String name;
    private double batteryLevel;

    // 建構子：用於創建物件
    public Robot(String name) {
        this.name = name;
        this.batteryLevel = 100.0;
    }
```

```
    // 方法：描述物件的行為
    public void move(double distance) {
        // 移動機器人的程式碼
        batteryLevel -= distance * 0.1;  // 移動會消耗電池
    }

    public double getBatteryLevel() {
        return batteryLevel;
    }
}
```

- 類別與物件的關係：
  - 類別是模板，物件是根據該模板創建的實例
  - 一個類別可以創建多個物件
  - 類比：「房屋設計圖」vs「實際建造的房屋」
- 類別設計原則：
  - 封裝：隱藏內部實現，只開放必要的介面
  - 一致性：相關功能放在同一類別
  - 合理分工：每個類別只負責單一領域功能

14：物件（Object）與實例化

- **標題：** 物件（Object）與實例化
- **內容：**
  - 什麼是物件？
    - 物件是類別的實例
    - 具有類別定義的屬性和方法
    - 每個物件都是獨立的，有自己的屬性值
  - 創建物件（實例化）：

```
// 使用「new」關鍵字創建物件
Robot astrobee = new Robot("Astrobee");
Robot bumble = new Robot("Bumble");

// astrobee和bumble是兩個不同的物件
// 它們有相同的方法，但可以有不同的屬性值
```

  - 訪問物件的方法和屬性：

```
// 呼叫物件的方法
astrobee.move(5.0);
```

```
// 獲取物件的屬性（通過方法）
double batteryLeft = astrobee.getBatteryLevel();  // 可能是99.5
double bumbleBattery = bumble.getBatteryLevel();  // 應該是100.0
```

- 在Kibo-RPC中常用的物件：

```
// 創建點物件
Point targetPosition = new Point(11.5, -7.5, 4.5);

// 創建方向物件
Quaternion orientation = new Quaternion(0, 0, 0.707, 0.707);

// 使用這些物件
api.moveTo(targetPosition, orientation, false);
```

- 為什麼需要物件：
  - 對現實世界的建模更直觀
  - 程式碼更有組織性和可重用性
  - 資料和行為緊密結合
  - 便於團隊合作開發

15：Java檔案結構與Package

- **標題**： Java檔案結構與Package
- **內容**：
  - Java檔案的基本結構：

```
// 1. Package宣告（必須是第一行程式碼）
package jp.jaxa.iss.kibo.rpc.sampleapk;

// 2. Import陳述句（引入需要的類別）
import java.util.ArrayList;
import org.opencv.core.Mat;
import gov.nasa.arc.astrobee.types.Point;

// 3. 類別宣告
public class YourService extends KiboRpcService {
    // 類別主體
}
```

  - Package的作用：
    - 組織和管理相關的類別
    - 避免命名衝突（不同package可以有同名類別）
    - 控制類別的存取權限

        - 提供更好的封裝性
    - Package命名慣例：
        - 使用「反向域名」模式
        - 全部小寫字母
        - 例如：`jp.jaxa.iss.kibo.rpc.sampleapk`
            - 代表日本宇宙航空研究開發機構（JAXA）
            - 國際太空站（ISS）
            - Kibo機器人程式設計挑戰賽（Kibo RPC）
            - 範例應用程式（Sample APK）
    - Kibo-RPC中的package結構：
        - 每個團隊應該創建自己的package
        - 例如：`jp.jaxa.iss.kibo.rpc.teamname`
        - 您的主要Service類別應位於此package中

16：Import陳述句

- **標題：** Import陳述句的用途
- **內容：**
    - Import的作用：
        - 告訴Java編譯器您打算使用的類別位置
        - 讓您可以使用簡化名稱，而非完整路徑
        - 例如：使用`Point`而非`gov.nasa.arc.astrobee.types.Point`
    - Import的語法：

```
// 引入單一類別
import org.opencv.core.Mat;

// 引入整個套件中的所有類別
import gov.nasa.arc.astrobee.types.*;
```

    - Kibo-RPC中常用的import：

```
// Astrobee控制API
import gov.nasa.arc.astrobee.types.Point;
import gov.nasa.arc.astrobee.types.Quaternion;
import gov.nasa.arc.astrobee.Result;

// OpenCV用於圖像處理
import org.opencv.core.Mat;
import org.opencv.imgproc.Imgproc;
import org.opencv.core.MatOfPoint;
```

```
// Java標準庫
import java.util.ArrayList;
import java.util.List;
```

- Import注意事項：
  - Import陳述句放在package宣告之後，類別宣告之前
  - 過多的wildcard import（*）可能導致名稱衝突
  - Java預設自動import `java.lang.*`（所以不需要import String、Integer等）

17：類別繼承（Extends）

- **標題：** 類別繼承（Extends）
- **內容：**
  - 繼承的概念：
    - 允許一個類別獲得另一個類別的屬性和方法
    - 建立「is-a」關係（子類別是父類別的一種特化）
    - 促進程式碼重用
  - 繼承的語法：

```
// 父類別（超類別）
public class Robot {
    // 父類別屬性和方法
    protected String name;

    public void move() {
        // 移動邏輯
    }
}

// 子類別（衍生類別）
public class Astrobee extends Robot {
    // 繼承了Robot的所有屬性和方法

    // 可添加自己特有的屬性和方法
    private boolean cameraActive;

    public void takePicture() {
        // 拍照邏輯
    }
}
```

  - 繼承特性：
    - 子類別獲得父類別的所有public和protected成員
    - 子類別可以添加新的成員
    - 子類別可以覆寫（override）父類別的方法

    - Java只支援單一繼承（一個類別只能有一個直接父類別）

- Kibo-RPC中的繼承：

```
public class YourService extends KiboRpcService {
    // YourService繼承自KiboRpcService
    // 因此獲得了KiboRpcService的所有功能

    @Override
    protected void runPlan1() {
        // 您的任務代碼
    }
}
```

18：方法覆寫（@Override）

- **標題：** 方法覆寫（@Override）
- **內容：**

    - 什麼是方法覆寫？

        - 子類別重新定義從父類別繼承的方法的實現
        - 方法簽名必須完全相同（名稱、參數、返回型態）
        - 使用`@Override`註解來明確表示覆寫意圖

    - 覆寫的語法：

```
public class Robot {
    public void move() {
        System.out.println("Robot moves slowly");
    }
}

public class Astrobee extends Robot {
    @Override
    public void move() {
        System.out.println("Astrobee flies in zero gravity");
        // 完全不同的實現
    }
}
```

    - `@Override`註解的作用：

        - 告訴編譯器這是一個覆寫方法
        - 如果父類別沒有匹配的方法，編譯器會報錯
        - 提高程式碼可讀性
        - 避免因打字錯誤導致創建新方法而非覆寫

    - Kibo-RPC中的方法覆寫：

```
public class YourService extends KiboRpcService {
    @Override
    protected void runPlan1() {
        // 您的任務實現代碼
        api.startMission();

        // 移動到指定位置
        Point point = new Point(10.9, -9.8, 4.3);
        Quaternion quaternion = new Quaternion(0, 0, -0.707, 0.707);
        api.moveTo(point, quaternion, false);

        // 獲取並處理圖像
        Mat image = api.getMatNavCam();
        // 處理image...
    }
}
```

19：Android應用程式結構

- **標題：** Android應用程式結構
- **內容：**
  - Android應用的組件：
    - **Activity**：有使用者界面的單一畫面
    - **Service**：在背景執行的長時間操作
    - **ContentProvider**：應用間共享數據
    - **BroadcastReceiver**：接收系統或應用的廣播
  - Kibo-RPC中的主要組件：
    - **MainActivity**：應用程式的使用者界面入口
    - **YourService**：執行機器人控制邏輯的Service
  - MainActivity的作用：
    - 提供使用者界面（若有）
    - 啟動YourService
    - 顯示任務狀態和結果
    - Kibo-RPC中，大部分操作在Service中進行，MainActivity較為簡單
  - YourService的作用：
    - 繼承自KiboRpcService
    - 包含實際的機器人控制邏輯
    - 實現`runPlan1()`方法，其中包含您的任務代碼
    - 在這裡使用Astrobee API控制機器人
  - Kibo-RPC的啟動流程：

1. 系統啟動MainActivity
2. MainActivity啟動YourService
3. YourService被系統調用，執行`runPlan1()`方法
4. 您的程式碼控制Astrobee完成任務

- 修改哪個部分？
  - 大多數情況下，您只需修改`YourService.java`中的`runPlan1()`方法
  - 通常不需要修改MainActivity或其他部分
  - 專注於實現機器人的任務邏輯

Java程式設計基礎 - Kibo-RPC教學 （第21-30張）

20：存取修飾詞 - Public

- **標題：** 存取修飾詞 - Public
- **內容：**
  - Java中的存取修飾詞控制類別、方法和變數的可見性
  - `public` 修飾詞：
    - 最開放的存取層級
    - 允許從任何地方訪問該成員
    - 語法：

```
public class Robot {
    public String name;

    public void move() {
        // 移動邏輯
    }
}
```

  - `public` 的使用場景：
    - 需要被外部類別廣泛使用的API方法
    - 代表類別主要功能的方法
    - 需要被其他套件使用的類別
  - 在Kibo-RPC中的應用：
    - `YourService` 類別通常宣告為 `public`
    - API提供的方法大多為 `public`，例如：

```
// 這些都是public方法，可以直接調用
api.startMission();
```

```
api.moveTo(point, quaternion, false);
Mat image = api.getMatNavCam();
```

- 設計建議：
  - 不要讓所有成員都是 `public`（破壞封裝性）
  - 方法通常比變數更適合設為 `public`
  - 只將必要的介面設為 `public`

21：存取修飾詞 - Private

- **標題：** 存取修飾詞 - Private
- **內容：**
  - `private` 修飾詞：
    - 最嚴格的存取層級
    - 僅允許在宣告它的類別內部訪問
    - 語法：

```
public class Robot {
    private double batteryLevel;  // 只能在Robot類別內部訪問

    private void rechargeBattery() {  // 只能在Robot類別內部呼叫
        batteryLevel = 100.0;
    }
}
```

  - `private` 的使用場景：
    - 類別的內部實現細節
    - 不希望外部直接訪問的資料
    - 只在類別內部使用的輔助方法
  - 封裝的概念：
    - 隱藏內部實現細節
    - 通過公開方法控制對私有資料的訪問
    - 保護資料的完整性

```
public class Robot {
    private double batteryLevel;  // 私有變數

    // 公開方法提供受控的訪問
    public double getBatteryLevel() {
        return batteryLevel;
    }
```

```
    public void setBatteryLevel(double level) {
        // 可以添加驗證邏輯
        if (level >= 0 && level <= 100) {
            batteryLevel = level;
        }
    }
}
```

- 在Kibo-RPC中的應用：
  - 輔助變數和方法通常設為 `private`
  - 例如，處理圖像識別的內部方法：

```
private Mat preprocessImage(Mat originalImage) {
    // 圖像預處理邏輯
    return processedImage;
}
```

23：存取修飾詞 - Protected與Default

- **標題：** 存取修飾詞 - Protected與Default
- **內容：**
  - `protected` 修飾詞：
    - 允許同一個套件內的類別和所有子類別訪問
    - 語法：

```
public class Robot {
    protected String model;  // 可被子類別訪問

    protected void initialize() {  // 可被子類別呼叫
        // 初始化邏輯
    }
}
```

  - `default`（無修飾詞）：
    - 也稱為「套件私有」（package-private）
    - 允許同一個套件內的所有類別訪問
    - 語法：

```
class Helper {  // 注意沒有public
    void utility() {  // 注意沒有存取修飾詞
        // 輔助功能邏輯
    }
}
```

- 存取修飾詞的可見性比較：
  - `public`: 所有地方可見
  - `protected`: 同套件和所有子類別可見
  - `default`（無修飾詞）: 同套件可見
  - `private`: 僅同類別可見
- 在Kibo-RPC中的應用：
  - `KiboRpcService` 中的一些方法是 `protected`，允許您的 `YourService` 覆寫它們：

```
@Override
protected void runPlan1() {
    // 您的任務程式碼
}
```

  - 自定義的輔助類別可以使用 `default` 存取層級，只在您的套件內使用

23：使用new關鍵字建立物件

- **標題：** 使用new關鍵字建立物件
- **內容：**
  - `new` 關鍵字的作用：
    - 創建類別的新實例（物件）
    - 為物件配置記憶體空間
    - 呼叫類別的建構子進行初始化
  - 基本語法：

```
類別名稱 變數名稱 = new 類別名稱(參數列表);
```

  - 示例：

```
// 創建不含參數的物件
ArrayList<String> items = new ArrayList<>();

// 創建含參數的物件
Point position = new Point(10.5, -7.2, 4.3);

// 先宣告變數，後創建物件
Quaternion orientation;
orientation = new Quaternion(0, 0, -0.707, 0.707);
```

- 建構子：
  - 特殊的方法，與類別同名
  - 在創建物件時自動呼叫
  - 用於初始化物件的狀態

```
public class Robot {
    private String name;

    // 建構子
    public Robot(String name) {
        this.name = name;
    }
}

// 使用建構子創建物件
Robot astrobee = new Robot("Astrobee");
```

- 在Kibo-RPC中常見的物件創建：

```
// 創建位置物件
Point point = new Point(10.9, -9.8, 4.3);

// 創建方向物件
Quaternion quaternion = new Quaternion(0, 0, -0.707, 0.707);

// 創建儲存容器
ArrayList<String> detectedItems = new ArrayList<>();
```

24：API概念介紹

- **標題**： API概念介紹
- **內容**：
  - 什麼是API？
    - API = Application Programming Interface（應用程式介面）
    - 預先定義好的方法集合，允許程式間溝通
    - 隱藏複雜實現細節，提供簡單使用介面
    - 就像遙控器一樣，無需了解電視內部電路，就能操作電視
  - API的優點：
    - 減少重複工作
    - 隱藏複雜度
    - 標準化操作
    - 提高程式碼可讀性
    - 允許系統分層

- API的類型：
  - 程式庫API（如Java標準庫）
  - 作業系統API
  - Web API（RESTful API等）
  - 硬體API（如控制機器人）
- Kibo-RPC中的API：
  - Astrobee API允許您控制太空站上的機器人
  - 隱藏了機器人運作的複雜細節
  - 提供了簡單易用的方法來完成任務
  - 通過 `api` 物件訪問，它是 `YourService` 中的一個實例變數

25：Kibo-RPC API的使用

- **標題：** Kibo-RPC API的使用
- **內容：**
  - 訪問Kibo-RPC API：
    - 在 `YourService` 類別中，已經定義了 `api` 實例變數
    - 不需要創建新的API物件，直接使用現有的 `api` 變數
    - 所有機器人控制方法都通過 `api` 物件呼叫
  - 常用API方法：
    - 任務控制：

```
api.startMission();  // 開始任務，初始化機器人
```

    - 移動控制：

```
// 移動到指定位置和方向
api.moveTo(point, quaternion, false);
```

    - 相機控制：

```
// 獲取導航相機圖像
Mat image = api.getMatNavCam();
```

    - 區域資訊：

```
// 設定區域中識別到的物品資訊
api.setAreaInfo(1, "item_name", 1);
```

- 任務完成：

```
// 拍攝目標物品照片並完成任務
api.takeTargetItemSnapshot();
```

- API方法返回值：
  - 有些方法返回資料（如 `getMatNavCam()`）
  - 有些方法返回狀態（如 `startMission()` 返回布林值）
  - 有些方法不返回值（void方法）
- 在Kibo-RPC任務中的API使用流程：
  1. 呼叫 `api.startMission()` 開始任務
  2. 使用 `api.moveTo()` 移動到各個區域
  3. 使用 `api.getMatNavCam()` 獲取圖像
  4. 使用 `api.setAreaInfo()` 報告識別結果
  5. 使用 `api.takeTargetItemSnapshot()` 完成任務

26：API呼叫實戰範例

- **標題：** API呼叫實戰範例
- **內容：**
  - 基本任務流程代碼：

```
@Override
protected void runPlan1() {
    // 步驟1: 開始任務
    api.startMission();

    // 步驟2: 移動到第一個區域
    Point point1 = new Point(10.9, -9.9, 5.2);
    Quaternion quaternion1 = new Quaternion(0, 0, -0.707, 0.707);
    api.moveTo(point1, quaternion1, false);

    // 步驟3: 拍攝並處理圖像
    Mat image1 = api.getMatNavCam();
    // 圖像處理邏輯...

    // 步驟4: 報告識別結果
    api.setAreaInfo(1, "emerald", 2);

    // 步驟5: 移動到下一個區域...繼續任務...
}
```

  - API錯誤處理：

```
// 檢查API呼叫的返回值
boolean missionStarted = api.startMission();
if (!missionStarted) {
    // 處理任務啟動失敗情況
}

// 使用try-catch處理潛在的異常
try {
    Result result = api.moveTo(point, quaternion, false);
    if (!result.hasSucceeded()) {
        // 處理移動失敗情況
        String errorMessage = result.getMessage();
        // 嘗試替代方案...
    }
} catch (Exception e) {
    // 處理異常情況
}
```

- API呼叫最佳實踐：
  - 在關鍵操作後檢查返回值
  - 為長時間操作添加適當的錯誤處理
  - 避免無限循環等危險模式
  - 保持代碼模塊化，便於調試
- 實戰提示：
  - 使用 `api.saveMatImage(image, "debug")` 保存中間圖像用於調試
  - 任務執行需要考慮容錯和恢復策略
  - 測試不同情境下的API行為

27：Kibo-RPC任務流程分析

- **標題**： Kibo-RPC任務流程分析
- **內容**：
  - 任務目標：
    - 從起始位置出發，巡視各個區域
    - 識別每個區域中的物品
    - 找到並拍攝目標物品
    - 完成任務
  - 任務挑戰：
    - 機器人定位與移動
    - 圖像識別
    - 路徑規劃
    - 時間管理

- 建議任務流程：

    1. **準備階段：**
        - 呼叫 `api.startMission()` 開始任務
        - 初始化必要的變數和資料結構
    2. **巡視階段：**
        - 依次移動到各個區域
        - 在每個區域拍攝並分析圖像
        - 記錄識別到的物品資訊
        - 使用 `api.setAreaInfo()` 報告識別結果
    3. **找出目標階段：**
        - 移動到宇航員位置
        - 報告巡視完成 `api.reportRoundingCompletion()`
        - 獲取目標物品資訊
        - 分析目標物品與已識別物品的關係
    4. **完成任務階段：**
        - 移動到目標物品所在區域
        - 拍攝目標物品照片 `api.takeTargetItemSnapshot()`

- 程式架構建議：
    - 使用模塊化設計，將各階段拆分為獨立方法
    - 建立錯誤恢復機制
    - 優化路徑規劃，考慮時間因素

28：實用程式技巧

- **標題：** 實用程式技巧
- **內容：**
    - 命名規範：
        - 類別名稱：大駝峰式（首字母大寫，如 `YourService`）
        - 方法與變數名稱：小駝峰式（首字母小寫，如 `startMission`）
        - 常數：全大寫加底線（如 `MAX_SPEED`）
        - 套件名稱：全小寫（如 `jp.jaxa.iss.kibo.rpc`）
    - 程式碼組織：
        - 相關功能放在同一個方法中
        - 長方法應拆分為多個較小方法
        - 使用有意義的變數名
        - 添加適當的註解
    - 常見錯誤與解決方案：

- 空指針異常（NullPointerException）：

```
// 避免
Mat image = api.getMatNavCam();
double value = processImage(image);  // 若image為null會出錯

// 改進
Mat image = api.getMatNavCam();
if (image != null) {
    double value = processImage(image);
}
```

- 索引越界異常（IndexOutOfBoundsException）：

```
// 避免
List<String> items = getItems();
String firstItem = items.get(0);  // 若列表為空會出錯

// 改進
List<String> items = getItems();
if (!items.isEmpty()) {
    String firstItem = items.get(0);
}
```

- 測試與調試技巧：
  - 使用 `api.saveMatImage()` 保存中間圖像
  - 模塊化測試，逐步驗證
  - 考慮邊界情況
  - 記錄關鍵變數值

29：示例程式分析

- **標題**： 示例程式分析
- **內容**：
  - 以下是一個完整的Kibo-RPC任務示例：

```
@Override
protected void runPlan1() {
    // 1. 開始任務
    api.startMission();

    // 2. 巡視第一個區域
    moveToArea(1);
    String item1 = identifyItem(1);

    // 3. 巡視第二個區域
```

```
        moveToArea(2);
        String item2 = identifyItem(2);

        // 4. 巡視第三個區域
        moveToArea(3);
        String item3 = identifyItem(3);

        // 5. 巡視第四個區域
        moveToArea(4);
        String item4 = identifyItem(4);

        // 6. 移動到宇航員位置報告完成
        moveToAstronaut();
        api.reportRoundingCompletion();

        // 7. 獲取目標物品資訊
        String targetItem = identifyTargetItem();
        api.notifyRecognitionItem();

        // 8. 找出目標物品位置
        int targetArea = findTargetItemArea(targetItem);

        // 9. 移動到目標物品位置
        moveToArea(targetArea);

        // 10. 拍攝目標物品並完成任務
        api.takeTargetItemSnapshot();
    }

    // 移動到指定區域
    private void moveToArea(int areaId) {
        Point point;
        Quaternion quaternion = new Quaternion(0, 0, -0.707, 0.707);

        switch (areaId) {
            case 1:
                point = new Point(10.9, -9.9, 5.2);
                break;
            case 2:
                point = new Point(10.9, -8.9, 4.8);
                break;
            case 3:
                point = new Point(10.9, -7.9, 4.8);
                break;
            case 4:
                point = new Point(10.9, -6.9, 5.2);
                break;
            default:
                return;
        }

        api.moveTo(point, quaternion, false);
    }
```

```
    // 識別區域中的物品
    private String identifyItem(int areaId) {
        Mat image = api.getMatNavCam();

        // 圖像處理邏輯（示例）
        String itemName = "unknown";
        int itemCount = 0;

        // 根據圖像識別結果設置物品名稱和數量
        // 這裡應該是您的圖像處理邏輯

        // 報告識別結果
        api.setAreaInfo(areaId, itemName, itemCount);

        return itemName;
    }

    // 移動到宇航員位置
    private void moveToAstronaut() {
        Point astronautPoint = new Point(11.2, -7.0, 5.0);
        Quaternion quaternion = new Quaternion(0, 0, 0.707, 0.707);
        api.moveTo(astronautPoint, quaternion, false);
    }

    // 識別目標物品
    private String identifyTargetItem() {
        Mat image = api.getMatNavCam();

        // 目標物品識別邏輯（示例）
        String targetItemName = "emerald";

        return targetItemName;
    }

    // 找出目標物品所在區域
    private int findTargetItemArea(String targetItem) {
        // 根據之前識別的結果找出目標物品所在區域
        // 這裡應該是您的邏輯
        return 2;  // 示例返回值
    }
```

- 重點分析：
  - 主方法 `runPlan1()` 概述了整個任務流程
  - 輔助方法處理特定任務，使代碼更模塊化
  - 使用適當的資料結構存儲識別結果
  - 包含錯誤處理和邏輯分支
- 改進空間：
  - 添加更多錯誤處理
  - 優化路徑規劃

- 改進圖像處理算法
- 使用更靈活的資料結構