COMSM1302 電腦架構概論
第十一週完整教材：Jack 語言與編譯器

Jack 語言 · 編譯 Jack · 類別的編譯 · 課程總結

依據 John Lapinskas（University of Bristol）課程講義編寫並延伸

2026 年 7 月

本教材的使用方式：本講義整合了第十一週四份投影片（11-1 The Jack language、11-2 Compiling Jack、11-3 Compiling Jack's classes、11-4 Summing up and looking forward）的全部內容，並補充 nand2tetris 第 10、11 章的相關細節。每章結尾附有「本章重點」整理；第 5 章為綜合練習題，附完整詳解。這是課程的最後一週：終於抵達抽象階梯的最頂端——**高階語言 Jack**，並學習把它編譯到 Hack VM。搭配前兩週的 VM 翻譯器與第八週的組譯器，我們就有了**一條完整的編譯鏈**：Jack → Hack VM → 組語 → 機器碼。注意：**Jack 語言本身不列入考試**——本課程在乎的不是用 Jack 寫程式，而是**編譯** Jack。

目錄

1　Jack 語言（The Jack Language）

1.1　Jack 的精神

討論 Jack 時最常出現的兩句話：

- 「**就像 C，但是……**」
- 「**為了（編譯的）簡單起見……**」

Jack 的每一個設計決策都圍繞一個目標：讓**學生能在一週內寫出它的編譯器**。語言的不便之處（稍後一一列出）全是刻意用「程式設計師多打幾個字」換取「編譯器少寫幾百行」。

1.2　變數、敘述與註解

變數與敘述

- 變數宣告像 C，但開頭加 `var` 關鍵字：`var int x;`、`var char a, b, c;`。
- 內建型別只有 `char`、`int`、`boolean`；函式庫另提供 `Array` 與 `String`。
- 與 C 不同，**宣告時不能初始化**：`var int x = 0;` 不合法。
- 為求簡單，所有變數必須**在函式開頭**宣告。
- 指定敘述像 C，但開頭加 `let`：`let x = 5;`。不支援 `+=`、`*=` 等複合指定。
- 註解與 C 相同（`//` 與 `/*...*/`），換行與空白被忽略（只用來分隔權杖）。

這些額外關鍵字（`var`、`let`）純粹是讓解析與編譯容易一點——BASIC 等早期語言的常見特徵：**看第一個權杖就知道這行是什麼敘述**。

1.3　函式

函式宣告像 C，但開頭加 `function` 關鍵字：

```
function int Main.max (int x, int y) {
    // 函式本體
}
```

無返回值的函式用 `void` 宣告。返回語法同 C（`return 42;` 或 `return;`），但**每個函式都必須以 `return` 結尾**。函式呼叫也像 C，唯一例外：**只呼叫函式並丟棄返回值**的一行必須以 `do` 開頭——`Main.print("Hello");` 不合法，`do Main.print("Hello");` 才合法。

1.4　運算式

運算式（expression）

在所有程式語言中，**運算式**是敘述中**會回傳值**的部分。字面值（`5`、`true`、`"Hello"`）是運算式，變數名也是。凡是能寫字面值的地方都能寫更複雜的運算式，且長度與複雜度**沒有上限**。`do`、`let`、`return` 後面都能接運算式，函式呼叫的引數也能是運算式。

支援 ✓	為簡單起見不支援 ✗
算術：`+`、`-`、`*`、`/`	模除 `%`；位移 `<<`、`>>`
邏輯：`&`、`|`、`~`（NOT）——兼作位元與邏輯運算	取址／解參考（單元 `*` 與 `&`）
比較：`=`（不是 `==`）、`>`、`<`	`++`、`--`；`!=`、`<=`、`>=`
字面值：整數、字串、`true`、`false`、`null`（= `false`）	運算式內的指定（賦值）
陣列下標 `[]`；變數；函式呼叫；括號 `()`	三元運算子 `?:` **沒有運算子優先順序**（無括號時）！`1+2*3` 可能是 7 也可能是 9

1.5　流程控制

Jack 支援與 C 相同的 `if-else` 與 `while`，但：

- **不支援 `else if`**——要寫成 `else { if (...) {...} }`（巢狀）；
- 不支援 `do...while` 與 `for`。

```
if (x > 5 & ~(y + f(x) = 7)) {
    // ...
} else { if (z = 2) {   // else if 不合法，巢狀才合法
    // ...
}}
while (x < 5 | Main.fibonacci(x) = 13) {
    // ...
}
```

1.6　型別

Jack **沒有轉型（cast）**。`int`、`char`、`boolean` 之間支援**隱含轉換**：`char` 依 Hack 字元集視為 `int`；`true` = −1、`false` = 0。例如：

```
var int x; var boolean y; var char z;
let x = -1; let y = x; let z = x + 71;
```

合法，且 y 為 `true`、z 為 `"F"`（$-1+71=70$，Hack 字元集的 F）。實作上「免費」取得：把 `char` ／ `boolean` 在記憶體中**一律存成 int**，然後**完全忽略型別資訊**。（認真做型別檢查會極度複雜——甚至有論文證明 Java 的精確型別檢查在理論上不可判定，除非接受檢查器可能不停機！）

1.7　一個「新」概念：類別（Classes）

Jack 用**類別**取代 C 的 struct。類別是物件導向程式設計（OOP，下學期）的核心，但幸運的是，Jack **缺少**讓類別比 struct 複雜的所有（眾多）特性。Jack 類別與 C struct 唯一的實質差別：

- C 中與 struct 相關的函式與 struct 定義**分離**（甚至可以在不同檔案）；
- Jack 中與類別相關的函式**包含在**類別定義之內。

1.7.1　類別變數：field 與 static

- `field` 變數 = C 的一般 struct 成員；
- `static` 變數為**該類別所有成員共享**（C 的 struct 沒有這個）；
- 類別變數用一般語法宣告（`var Foo myFoo;`），所有類別變數必須在**類別開頭**宣告；
- Jack **不支援** `myFoo.x` 語法存取欄位——改用**方法**。

1.7.2　方法（Methods）

方法

方法是「屬於」類別的特殊函式，只能從該類別的 **特定實例**呼叫。方法定義內部：類別變數的欄位**像區域變數一樣**直接使用（取代 C 的 `myFoo.x`）；類別變數本身可用 `this` 關鍵字存取；`methodCall()` 解讀為 `this.methodCall()`。

C 版本：

```
void twiddleFoo(struct Foo *abc) {
    abc->x = abc->x + abc->y;
    abc->y = abc->y - 1;
}
twiddleFoo(myFoo);
twiddleFoo(myOtherFoo);
```

Jack 版本：

```
class Foo {
    method void twiddle() {
        let x = x + y;
        let y = y - 1;
    }
}
// 呼叫端：
do myFoo.twiddle();
do myOtherFoo.twiddle();
```

第一次呼叫時 `this` 是 `myFoo`、第二次是 `myOtherFoo`——正如 C 版的 `abc`。

1.7.3 建構子（Constructors）

建構子是用來**建立新類別變數**的特殊函式。執行期，建構子在呼叫開始時自動在**堆積**上建立新變數，`this` 指向它，且建構子應以 `return this;` 結尾：

```
class Foo {
    field int x, y;
    constructor Foo newFoo(int a) {
        let x = 5;
        let y = a;
        return this;
    }
}
// 呼叫端：
let myFoo = Foo.newFoo(42);
```

這對應 C 的 `malloc(sizeof(Foo))` + 欄位初始化 + 回傳指標。呼叫建構子與函式時必須帶類別名前綴（`Foo.newFoo()`，不能只寫 `newFoo()`）。注意：Jack 的類別配置在堆積上，所以**必須像 C 的 malloc 指標一樣手動釋放**——慣例是寫一個 `dispose` 方法，內部呼叫 `Memory.deAlloc(this)`。（投影片的兩個範例其實都漏了釋放——都會漏記憶體！）

1.8 [] 語法與記憶體後門

Jack 中**所有類別型變數都存在堆積**，只有 `int` ／ `char` ／ `boolean` 存在堆疊。實際上類別型變數就是**指標**——把 `Foo` 型變數當 `int` 讀，會看到欄位所在的堆積位址。運算式 `myObject[i]` 的意思是：取 `myObject` 的位址、加 i、回傳該處的值。`Array` 與 `String` 類別的實作保證位址就是陣列／字串的第一個元素，所以語法行為符合直覺。（這是純 Jack 的做法——C 的 struct 放堆疊！）

這也給了我們一個**記憶體後門**：

```
var int x;
let x = 16384;
let x[0] = 0;   // 把 RAM[0x4000]（螢幕第一個字）設為 0
```

1.9 多檔案編譯與範例程式

每個 Jack 檔案含**一個**類別宣告（`Foo.jack` 含類別 `Foo`）。**一切都必須在類別內**；不屬於任何類別的程式碼放 `Main` 類別。程式從 `Main.main()` 開始（類比 C 的 `main`）。每個 Jack 檔案**分開**編譯成 Hack VM，再由 VM 翻譯器合併、組譯器產出機器碼。Jack 檔案可以使用其他檔案定義的類別——編譯器**直接假設**它們存在（省掉 C 的 makefile，代價是更差的錯誤處理）。

投影片以 Nisan and Schocken 的 `Average` 程式收尾：讀入若干數字、用 `Array.new` 配置陣列、`while` 迴圈累加、輸出平均值——其中 `Keyboard.readInt`、`Output.printString` 等都來自標準函式庫（寫編譯器時不需要管它們！）。

本章重點

- Jack = 「像 C，但為編譯簡單而處處讓步」；語言細節不考，**編譯**才是重點。
- `var` ／ `let` ／ `do` 前綴關鍵字 ⇒ 第一個權杖決定敘述種類。
- 無運算子優先順序（`1+2*3` 不確定）、無 `else if`、無 `for`、無轉型、宣告不能初始化。
- 類別 = struct + 內建函式；`field` ／ `static` ／方法／建構子；物件在堆積、手動 `dispose`。
- `myObject[i]` = $RAM[\texttt{myObject} + i]$ ——兼作陣列語法與記憶體後門。

2　編譯 Jack（Compiling Jack）

2.1　詞法分析（Lexing）

Jack 的詞法分析毫無驚喜，技術與之前完全相同。權杖如下：

類別	內容
關鍵字	`class`、`constructor`、`function`、`method`、`field`、`static`；`int`、`char`、`boolean`、`void`；`var`、`let`、`do`、`if`、`else`、`while`、`return`；`true`、`false`、`null`、`this`
符號	`+ - * / & | ~ < > =`；`{ } [ ] ( ) . , ;`
整數字面值	十進位 0…32767
字串字面值	雙引號夾住、不含換行與額外雙引號的字元序列
識別字	字母、數字、底線組成，不以數字開頭、非關鍵字

兩個值得注意的地方：

- 識別字的定義比 Hack VM ／組語**更嚴格**——這讓 `myVar+4` 不加空格也能無歧義地切出權杖；
- **換行不是權杖**！跟 C 一樣只是空白。（對照：組語與 VM 都以換行分隔指令。）

2.2　Jack 的文法（EBNF）

回憶我們的 EBNF 方言：`{}` 表示「重複任意次（含零次）」、`[]` 表示「可有可無」。

結構層文法

⟨*class*⟩ ::= ‘`class`’, identifier, ‘`{`’, {⟨*classVarDec*⟩}, {⟨*subroutineDec*⟩}, ‘`}`’

⟨*classVarDec*⟩ ::= (‘`static`’ | ‘`field`’), ⟨*type*⟩, identifier, {‘`,`’, identifier}, ‘`;`’

⟨*type*⟩ ::= ‘`int`’ | ‘`char`’ | ‘`boolean`’ | identifier

⟨*subroutineDec*⟩ ::= (‘`constructor`’ | ‘`function`’ | ‘`method`’), (‘`void`’ | ⟨*type*⟩), identifier, ‘`(`’, ⟨*parameterList*⟩, ‘`)`’, ⟨*subroutineBody*⟩

⟨*parameterList*⟩ ::= [⟨*type*⟩, identifier, {‘`,`’, ⟨*type*⟩, identifier}]

⟨*subroutineBody*⟩ ::= ‘`{`’, {⟨*varDec*⟩}, ⟨*statements*⟩, ‘`}`’

⟨*varDec*⟩ ::= ‘`var`’, ⟨*type*⟩, identifier, {‘`,`’, identifier}, ‘`;`’

注意所有變數都宣告在類別開頭（⟨*classVarDec*⟩）或函式開頭（⟨*varDec*⟩）——建符號表時，**在需要產生任何涉及變數的程式碼之前，變數的全部資訊都已到手**！

敘述層文法

⟨*statements*⟩ ::= {⟨*letStatement*⟩ | ⟨*ifStatement*⟩ | ⟨*whileStatement*⟩
| ⟨*returnStatement*⟩ | ⟨*doStatement*⟩}
⟨*letStatement*⟩ ::= ‘`let`’, identifier, [‘`[`’, ⟨*expression*⟩, ‘`]`’], ‘`=`’, ⟨*expression*⟩, ‘`;`’
⟨*ifStatement*⟩ ::= ‘`if`’, ‘`(`’, ⟨*expression*⟩, ‘`)`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’,
[‘`else`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’]
⟨*whileStatement*⟩ ::= ‘`while`’, ‘`(`’, ⟨*expression*⟩, ‘`)`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’
⟨*doStatement*⟩ ::= ‘`do`’, ⟨*subroutineCall*⟩, ‘`;`’
⟨*returnStatement*⟩ ::= ‘`return`’, [⟨*expression*⟩], ‘`;`’

注意 ⟨*ifStatement*⟩ 與 ⟨*whileStatement*⟩ 中 ⟨*statements*⟩ 的**遞迴**。⟨*statements*⟩ 內不能宣告新變數（宣告只出現在 ⟨*varDec*⟩ ／ ⟨*classVarDec*⟩），所以**不必追蹤獨立的作用域**。

運算式層文法

⟨*expression*⟩ ::= ⟨*term*⟩, {(‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`&`’ | ‘`|`’ | ‘`<`’ | ‘`>`’ | ‘`=`’), ⟨*term*⟩}
⟨*term*⟩ ::= integer literal | string literal | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’
| identifier, [‘`[`’, ⟨*expression*⟩, ‘`]`’] | ‘`(`’, ⟨*expression*⟩, ‘`)`’
| ((‘`-`’ | ‘`~`’), ⟨*term*⟩) | ⟨*subroutineCall*⟩
⟨*subroutineCall*⟩ ::= identifier, [‘`.`’, identifier], ‘`(`’, ⟨*expressionList*⟩, ‘`)`’
⟨*expressionList*⟩ ::= [⟨*expression*⟩, {‘`,`’, ⟨*expression*⟩}]

看看運算式**遞迴得多兇**！而這種遞迴是捕捉 `Math.sqrt((x+y)*Main.fibonacci(z))` 這類運算式 **絕對必要**的。我們拖延了很久，但現在**必須**用解析樹來處理它們——也就是終於把**解析**與 **程式碼產生**分開。幸好 Jack 是 **LL(2)** 語言（而且接近 LL(1)），過程不會太痛苦。

2.3　用 XML 儲存解析樹

解析趟與後續程式碼產生趟之間，解析樹要存在哪？理想上要：**人類可讀**（方便除錯）、**夠標準**（讀寫檔案不費事）、**夠彈性**（不必整棵樹載入記憶體就能瀏覽）。nand2tetris 正確地指出 **XML** 三項全中——多數語言都有現成的標準函式庫。當然，C 不是「多數語言」……所以課程把第 8–10 週的權杖程式碼改造成陽春的 XML 處理函式庫放進骨架；詞法器也用同一函式庫把權杖輸出成 XML 格式。

2.4 一個（相當）通用的 LL(2) 解析器

解析器契約

解析權杖串列時，維護兩個指標：`current`（第一個尚未解析的權杖）與 `lookahead`（第二個）。每個非終端符號有自己的函式（`parse_term`、`parse_statements`……）。`parse_abc` 的契約：

- 呼叫開始時，`current` 是一個 ⟨*abc*⟩ 非終端符號的**第一個**權杖；
- 產生該非終端符號的**全部 XML**（用 `write_tag` 寫開閉標籤、`copy_tag` 複製權杖）；
- 返回時 `current` 指向 ⟨*abc*⟩ **結束後的第一個**權杖。

達成的手段：**遞迴**。唯一需要 `lookahead` 的地方是解析 ⟨*term*⟩：若 ⟨*term*⟩ 的第一個權杖是識別字，它可能是變數，也可能是 ⟨*subroutineCall*⟩ 的函式名——看下一個權杖是不是 '(' 或 '.' 才能分辨。

解析追蹤範例

投影片逐步解析了：

```
while (count < 100) {
    let count = count + 1;
}
```

關鍵步驟（縮排表示遞迴深度）：

1. `parse_while_statement`：寫 `<whileStatement>`，複製 `while`、`(`；
2. `parse_expression`：寫 `<expression>`；
3. `parse_term`：由 `lookahead`（是 `<`，不是 `(`／`.`）判定 `count` 不是 ⟨*subroutineCall*⟩，複製識別字；`current` 不是 `[`，⟨*term*⟩ 結束；
4. `current` 是 `<`，運算式未完：複製 `<`，再遞迴 `parse_term` 處理 `100`（整數字面值直接收尾）；
5. `current` 是 `)`，不是運算子 ⇒ ⟨*expression*⟩ 結束、寫閉標籤返回；
6. 複製 `)`、`{`，進入 `parse_statements` → `parse_let_statement`（同樣模式）……
7. `current` 是 `}`，不是 let/while/if/do/return ⇒ ⟨*statements*⟩ 結束，複製 `}`、寫閉標籤、返回。

每一步只看 `current`（偶爾加 `lookahead`）就能決定動作——這就是 LL(2) 的實際體感。

2.5 從解析樹到 AST

我們的文法已經比 Nisan and Schocken 原版刪掉許多多餘的非終端符號（如 ⟨*op*⟩、⟨*varName*⟩）。建解析樹時，還應把 ⟨*type*⟩ 替換成它的定義。更進一步：`if`、`{}`、`()` 這些權杖對「把權杖串變成

⟨*ifStatement*⟩」有用，但**每個** ⟨*ifStatement*⟩ 都長一樣——把它變成程式碼時真的需要這些節點嗎？這正是 **AST（抽象語法樹）**的用意，且沒有唯一正解。建議做法：先寫**包含解析樹每個節點**的程式碼、用 `diff` 對照提供的 XML 輸出測試，再考慮要從 AST 略去什麼（把 `copy_tag` 換成 `advance_tag`）。

2.6　語意分析：變數映射與符號表

Jack 變數 → Hack VM 記憶體段

Jack 宣告	VM 段	宣告位置
`var`	`local`	⟨*subroutineBody*⟩ 開頭的 ⟨*varDec*⟩
函式參數	`argument`	⟨*subroutineDec*⟩ 的 ⟨*parameterList*⟩
`static`	`static`	⟨*class*⟩ 開頭的 ⟨*classVarDec*⟩
`field`	`this`	⟨*class*⟩ 開頭的 ⟨*classVarDec*⟩

`that`：讓「`[]` 與 field 並用」的運算式順利編譯（this 給欄位、that 給 `[]`）
`temp`：編譯單一敘述時的暫存；`pointer` ／ `constant` 的用途已明

Jack 被非常小心地設計成**不需要獨立一趟**來建符號表或做其他語意分析。解析之後直接進入程式碼產生，從頭到尾掃一遍檔案、邊讀 XML 邊產出程式碼。處理 ⟨*subroutineDec*⟩ 時，遇到開標籤就：

1. 為該副程式建一張**新符號表**；
2. 若是**方法**，先加一筆 argument 項目給 `this`、位移 0（原因見第 3 章！）；
3. 為 ⟨*parameterList*⟩ 的每個變數加一筆 argument 項目；
4. 為 ⟨*subroutineBody*⟩ 每個 ⟨*varDec*⟩ 加 local 項目；
5. 用符號表為 ⟨*statements*⟩ 產生程式碼；
6. 到 ⟨*subroutineBody*⟩ 結尾時**釋放**符號表（不再需要）。

每筆符號表項目包含：**變數名**（查詢鍵）、**型別**（字串）、**種類**（local ／ argument……）、**位移**（貪婪分配）。例如 Nisan and Schocken 圖 11.2 的表把 `this` 映射到 `argument 0`、`other` 到 `argument 1`、`dx` 到 `local 0`、`dy` 到 `local 1`。

2.7　程式碼產生：遞迴的真正威力

程式碼產生使用與解析**非常相似**的遞迴結構。

compile_while

回憶 ⟨*whileStatement*⟩ ::= ‘`while`’, ‘`(`’, ⟨*expression*⟩, ‘`)`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’，而你從第 5 週就知道怎麼用 goto 與標籤做 while 迴圈！`compile_while` 可以：

1. 產生兩個未用過的 VM 標籤，如 `while_start_4` 與 `while_end_4`；
2. 輸出 `label while_start_4`；

3. 前進到 ⟨*expression*⟩，呼叫 `compile_expression`——輸出「把迴圈條件的結果推上堆疊」的 VM 碼；
4. 輸出 `not` 與 `if-goto while_end_4`；
5. 前進到 ⟨*statements*⟩，呼叫 `compile_statements` 輸出迴圈本體；
6. 輸出 `goto while_start_4` 與 `label while_end_4`；
7. 前進越過 `}` 與閉標籤，返回。

compile_expression

回憶 ⟨*expression*⟩ ::= ⟨*term*⟩, {(op), ⟨*term*⟩}。`compile_expression` 可以：

1. 呼叫 `compile_term`——輸出「把第一個 ⟨*term*⟩ 的結果推上堆疊」的 VM 碼；
2. 只要還有 ⟨*term*⟩：存下運算子、前進到下一個 ⟨*term*⟩、呼叫 `compile_term`，再輸出對堆疊頂兩值執行該運算的 VM 碼——`+` 輸出 `add`，`*` 輸出 `call Math.multiply 2`（Hack VM 沒有乘法指令!）；
3. 返回。

記住 Jack 沒有運算子優先順序（除非加括號），所以這種**由左至右**的做法不會破壞任何東西——這正是 `()` 只作為 ⟨*term*⟩ 一部分出現的原因。

到此，完整 Jack-to-Hack 編譯器只剩一塊拼圖：**類別型變數與方法**的程式碼產生——見下一章。

本章重點

- 詞法：識別字比 VM 嚴格（可無空格切分）、換行只是空白。
- 變數全部宣告在開頭 ⇒ 建符號表不需獨立一趟；⟨*statements*⟩ 不含宣告 ⇒ 不需作用域堆疊。
- 運算式的遞迴文法逼出真正的解析樹（XML 儲存）；Jack 是 LL(2)——只有 ⟨*term*⟩ 開頭的識別字需要 lookahead。
- 變數映射：var→local、參數 →argument、static→static、field→this。
- 程式碼產生 = 對樹遞迴：while 用標籤對 + `not`/`if-goto`；運算式由左至右、`*` 呼叫 `Math.multiply`。

3 編譯 Jack 的類別

3.1 物件的運作方式

物件（object）

類別的實例稱為**物件**。在 Jack 中，**每個物件都是一個指標式陣列**。對任何類別 `Foo`：

```
var Foo myFoo;
// （初始化 myFoo 的程式碼）
do Output.printInt(myFoo);   // 印出 myFoo 存放的 RAM 位址！
```

若 `Foo` 依序定義欄位 `x`、`y`、`z`：`myFoo.x` 存於 $RAM[\texttt{myFoo}]$、`myFoo.y` 存於 $RAM[\texttt{myFoo}+1]$、`myFoo.z` 存於 $RAM[\texttt{myFoo}+2]$。因此 `myFoo[i]` 求值為 $RAM[\texttt{myFoo}+i]$ 是自然的——本例中 `myFoo[2]` 就是 `myFoo.z`。

所有物件欄位都配置在**堆積**；指標本身像一般 `var` 一樣存在**堆疊**。（與 C 的 struct 非常相似，除了 C 的 struct 可以放堆疊、且要處理不同欄位大小！）

3.2 副程式的期望行為

類別可含函式、方法與建構子，合稱**副程式 (subroutines)**（不尋常的術語——通常 subroutine 指 void 函式——但沿用 Nisan and Schocken）。所有副程式 `myClass.mySub` 應該：

- 編譯**呼叫**時：輸出把各引數（⟨*expression*⟩）推上堆疊的 VM 碼，接著輸出 `call myClass.mySub`；
- 編譯 ⟨*parameterList*⟩ 與 ⟨*varDec*⟩ 時：建符號表，用它把變數名換成編號的 local ／ argument；
- 編譯**返回**時：把返回的 ⟨*expression*⟩ 推上堆疊（無返回值就推**假值**），再輸出 `return`；
- 編譯 ⟨*doStatement*⟩ 時：⟨*subroutineCall*⟩ 之後 **務必** `pop temp 0`——丟棄無人要的返回值，避免堆疊上的「記憶體洩漏」。

3.3 三種副程式的差異

函式可以無視宿主類別（除了 static 變數）。但建構子與方法都關聯到其類別的**當前物件（current object）**：

- **建構子**被呼叫時自動**建立**當前物件——用 `Memory.alloc` 在堆積配置適當大小的段（重點是在呼叫結尾把當前物件返回）；
- **方法**通常用 `myVar.myMethod()` 語法呼叫（而非 `myClass.myMethod()`），以 `myVar` 為當前物件。

在方法與建構子的本體內：`this` 求值為當前物件；宿主類別的欄位 `x` 解讀為「`this` 的 `x`」；宿主類別的方法可用 `myMethod()` 呼叫（解讀為 `this.myMethod()`）。這對之後的 OOP 至關重要；在 Jack 的脈絡裡，它只是讓你能寫 `myToken.write(output)` 而不是 `write_token(myToken, output)`。

（附註：Jack **不**支援 C 式的 `myObject.myField` 欄位存取——要編譯它，必須在編譯某檔案時取得**另一個檔案**的類別欄位資訊，那需要跨所有檔案的完整語意分析趟——不是不可能，但很煩。）

3.4 類別符號表與 this 的角色

與副程式符號表一樣，我們只在**類別自身的程式碼內**需要知道 field ／ static 變數存在哪。遇到 ⟨*class*⟩ 開標籤：建新符號表 → 為每個 ⟨*classVarDec*⟩ 的變數加項目（field 與 static **分開編位移**）→ 在其後每個 ⟨*subroutineDec*⟩ 的程式碼產生中使用 → 遇閉標籤釋放。

this 段的關鍵不變式

我們終於用上 Hack VM 的 `this` 段！**不變式：`this 0` 永遠存放在當前物件指向的位址**。只要維持它，Jack 的「第 i 個欄位」就永遠映射到 VM 的 `this i`。

編譯 ⟨*subroutineCall*⟩ 時：

- 僅方法：把當前物件推上堆疊作為**新的第一個引數**（並加進符號表），再編譯其餘 ⟨*expressionList*⟩；`call` 的引數數量相應 +1；
- 分辨方法呼叫與其他呼叫：看有沒有 '.'、以及 '.' 左邊的識別字**是不是變數**。

編譯 ⟨*subroutineDec*⟩ 時：

- 方法：把 `pointer 0` 設為 `argument 0`（即呼叫者傳來的當前物件）；
- 建構子：呼叫 `Memory.alloc` 配置新物件（大小由類別符號表算出），把 `pointer 0` 設為基底位址；
- 兩者：在副程式本體中**避免**再動 `pointer 0`！

	函式	建構子	方法
呼叫語法	`myClass.mySub(a,b)`		`myVar.mySub(a,b)` 或 `mySub(a,b)`
呼叫時	一般行為		把 `myVar` 加為 argument 0
開始時	一般行為	this 基底 ← 新物件	this 基底 ← `myVar`
本體中	一般行為	欄位讀作 this 的欄位；`myMethod(a)` 讀作 `this.myMethod(a)`	
返回時	一般行為（建構子應永遠 `return this`）		

3.5 編譯 ⟨*term*⟩

`compile_term` 的目標：產生「求出 ⟨*term*⟩ 的值、留在堆疊頂」的 VM 碼。整數字面值 85 → `push constant 85`；括號運算式 → 呼叫 `compile_expression`。最難的情況是**識別字**與**字串字面值**。識別字：查類別與副程式符號表——

種類（位移 i）	輸出	說明
argument	`push argument i`	
var	`push local i`	
static	`push static i`	全類別物件共享，函式內也合法
field	`push this i`	合法 Jack 必在方法／建構子內；當前物件在 `pointer 0`，物件是陣列、第 i 個欄位在位置 i

若識別字**同時**出現在類別表與副程式表，**以副程式表優先**（區域遮蔽）。

3.6 字串字面值：官方做法與它的漏洞

官方 nand2tetris 做法：用 `String.new` 建一個上限長度合適的新 String → 用一連串 `String.appendChar` 填入字元（Hack 字元集對齊 ASCII，char 直接當 int 用）→ 把新 String 的位址推上堆疊。問題解決了，對吧？

有人在迴圈裡呼叫 `Output.printString("Uh-oh!")` 會發生什麼事？ 字面值的程式碼**每圈都跑一次**……每次都建新 String……每次都配置在堆積……然後**永遠沒人釋放**。**Jack 的「Hello, world!」自帶記憶體洩漏！**

該修嗎？ 要先講清楚「修」是什麼意思。某種意義上它**沒壞**——語言照規格運作；自動釋放字面值會**弄壞既有的 Jack 程式**（和測試腳本）。更嚴肅地說，看看 C 就知道沒有簡單答案：

- `char myString[] = "Hello";`——在**堆疊**上放一份副本，函式返回即消失，期間可修改。但從返回 `char *` 的函式 `return myString;` 就留下**懸空指標**。嗯。
- `char *myString = "Hello";`——程式開始時建一份**靜態**副本，指標指向它。但 `myString[0] = 'J';` 直接 **segfault**。糟糕。

講師的非官方 Hack：修改文法，讓 ⟨*letStatement*⟩ 與 ⟨*expressionList*⟩ 都能直接接受字串字面值。⟨*letStatement*⟩ 用官方做法（明確建立指標的人自己負責 `String.dispose`）；⟨*expressionList*⟩（函式引數）則**自動釋放**：

- `compile_expression_list` 把字串字面值引數的清單回傳給 `compile_subroutine_call`；
- `call` 之後，引數仍留在堆疊上（目前堆疊指標**之上**）——取回這些字面值、逐一呼叫 `String.dispose`；
- 陷阱：**第一個**引數會被返回值覆寫（即使函式是 void！）——所以若第一個引數是字串字面值，就把它**再推一次**當作額外的最後一個引數（`call` 的引數計數相應增加），釋放時改釋放最後一個。

本章重點

- 物件 = 堆積上的指標式陣列；第 i 個欄位在 $RAM[\text{物件位址} + i]$。

- 副程式 = 函式 + 方法 + 建構子；`do` 敘述後 `pop temp 0`、void 返回推假值。
- 不變式：`this 0` = 當前物件位址。方法：呼叫端把物件推為 argument 0、定義端 `pointer 0` ← `argument 0`；建構子：`Memory.alloc` + `pointer 0` ← 基底。
- ⟨*term*⟩ 識別字查表：argument/local/static/this i；副程式表優先於類別表。
- 字串字面值官方做法會洩漏記憶體（「Hello, world!」都漏！）；C 的兩種字串也各有陷阱；講師的修法 = 引數位置自動 dispose + 第一引數重推的技巧。

4 總結與展望（Summing Up）

4.1 知道我們不知道什麼

課程**沒有**做的事（每一項都可以是暑期專題!）：

- 用 **HDL**（如 Verilog）描述遠更複雜的電路、減少人工接線錯誤；
- 學更進階的組語（如 **MIPS**）並在低功耗環境開發；
- 硬體能力：進階運算、**中斷**、管線化、快取；
- 支援**多行程**的作業系統；
- 更好的編譯器：**型別檢查**、程式驗證、**程式碼最佳化**；
- 更多語言特性：真正的**物件導向**、**並行**；
- 網路連線（Twitch plays Tetris?）。

編譯器類專題找程式語言研究群；硬體／ OS 類找（系統）資安與 HPC 研究群。

4.2 我們做到了什麼

但別忘了我們**做到**的事——完整的四層編譯與建造：

1. **把 C 風格語言（Jack）編譯到堆疊機**：用文法與遞迴把複雜運算式轉成長串後綴式指令；用指標式陣列實作使用者自訂型別；用不洩漏的演算法配置與釋放堆積記憶體；用符號表管理不同作用域的變數；把 while 等複雜流程敘述化為簡單的 goto。
2. **把堆疊機編譯到組語**：用裸組語實作堆疊本身；用堆疊實作函式呼叫；把虛擬記憶體段映射回實體記憶體（堆疊與堆積皆然）；合併多檔案以支援函式庫。
3. **把組語編譯到原生機器碼**：把標籤映射到 ROM 位址；貪婪配置變數到指定記憶體區；用記憶體映射 I/O 寫螢幕、讀鍵盤；理解指令集本身及其設計理由。
4. **從零建造執行機器碼的電腦**：R-S 閂鎖 → D 正反器 → 暫存器與記憶體；用多工器／解多工器在元件間路由指令；用簡單邏輯閘實作複雜算術——一路回到卑微的 NAND 閘、真值表與布林代數。

還記得開場演講裡那些真的用麵包板與 NAND 晶片蓋出來的 Hack 電腦嗎？**你現在有能力做出它們了。真正的從零開始。**恭喜完成課程，考試順利！

十一週的抽象階梯（全景回顧）

層級	本課程對應的週次
高階語言（Jack）	第 11 週
中介表示法（Hack VM）	第 9–10 週
組語（Hack assembly）	第 5、8 週
指令集架構（Hack ISA）	第 7 週
微架構（Hack CPU）	第 7 週
元件（ALU、暫存器、RAM）	第 2–3 週
邏輯閘（NAND 等）	第 1 週
電晶體（CMOS）	第 4 週
物理	（隔壁系）

5 綜合練習題（附詳解）

練習 1：Jack 語法判斷

下列各行是否為合法 Jack？不合法者說明原因。(a) `var int x = 0;` (b) `Main.print("hi");` (c) `let x += 1;` (d) `if (x = 5) {...} else if (y = 2) {...}` (e) `while (~(x > 10)) { let x = x + 1; }`

解答

(a) ✗——宣告時不能初始化，要拆成 `var int x; let x = 0;`。(b) ✗——丟棄返回值的呼叫必須加 `do`：`do Main.print("hi");`。(c) ✗——不支援複合指定，要寫 `let x = x + 1;`。(d) ✗——不支援 `else if`，要寫 `else { if (y = 2) {...} }`。(e) ✓——`~` 是 NOT、比較用單一 `=`、while 語法正確。

練習 2：運算子優先順序

(a) Jack 中 `1 + 2 * 3` 可能求值為哪些結果？為什麼？(b) 若用第 2.7 節的 `compile_expression`（由左至右），會得到哪個？寫出產生的 VM 碼。(c) 要保證得到 7，Jack 程式設計師該怎麼寫？

解答

(a) **7 或 9**。Jack 規格**沒有**運算子優先順序（沒有括號時），編譯器可自由選擇結合方式：$1 + (2 \times 3) = 7$ 或 $(1 + 2) \times 3 = 9$ 都符合規格。

(b) 由左至右：先算 $1 + 2$、再乘 3，得 **9**。

```
push constant 1
push constant 2
add
push constant 3
call Math.multiply 2
```

（注意 `*` 編譯成呼叫 `Math.multiply`——Hack VM 沒有乘法指令。）

(c) 加括號：`1 + (2 * 3)`。括號運算式是 ⟨*term*⟩，會先被完整求值。

練習 3：隱含型別轉換

執行下列 Jack 程式後，`y` 與 `z` 的值（以 16 位元 int 表示）各是多少？

```
var int x; var boolean y; var char z;
let x = 3;
let y = (x > 2);
let z = 65 + x;
```

解答

`y`：$x > 2$ 為 true，Jack 的 true $= -1$（`0xFFFF`）——所以 `y` $= -1$。（VM 層 `gt` 本來就推 `0xFFFF`，「隱含轉換」實際上什麼都不用做。）`z`：$65 + 3 = 68$，即字元 `"D"`（Hack 字元集對齊 ASCII）。記憶體中 `y`、`z` 都只是 int——編譯器完全忽略型別資訊。

練習 4：EBNF 對照

用 2.2 節的文法，指出解析 `let a[i+1] = f(x) - 2;` 時，(a) 這整行匹配哪個非終端符號及哪個選項；(b) `i+1`、`f(x)`、`f(x) - 2` 各匹配什麼；(c) 解析器在讀到哪個權杖時知道有 `[...]` 部分？

解答

(a) ⟨*letStatement*⟩，帶可選的 [‘`[`’, ⟨*expression*⟩, ‘`]`’] 部分。(b) `i+1` 是 ⟨*expression*⟩（⟨*term*⟩ `i`、運算子 `+`、⟨*term*⟩ `1`）；`f(x)` 是 ⟨*subroutineCall*⟩（也是一種 ⟨*term*⟩）；`f(x) - 2` 是 ⟨*expression*⟩。(c) 讀完識別字 `a` 後，`current` 指向 `[`——單靠 `current` 即可分辨（不需 lookahead）：是 `[` 就解析下標，是 `=` 就直接進右值。

練習 5：為什麼是 LL(2)？

(a) 解析 ⟨*term*⟩ 時，哪種開頭權杖需要看 `lookahead`？列出 lookahead 的三種情況與對應動作。(b) 為什麼其他敘述（let/if/while/do/return）都只需 LL(1)？

解答

(a) 開頭是**識別字**時。看 `lookahead`：

- `(` 或 `.` ⇒ 這是 ⟨*subroutineCall*⟩（函式名或類別/變數名開頭）；
- `[` ⇒ 帶下標的變數（identifier + `[expression]`）；
- 其他 ⇒ 純變數。

（嚴格說 `[` 的情況用 `current` 也能處理——真正**必須** lookahead 的是區分變數與 ⟨*subroutineCall*⟩。）

(b) 因為 Jack 刻意設計成**每種敘述由第一個關鍵字唯一決定**：看到 `let` 就是 ⟨*letStatement*⟩、看到 `while` 就是 ⟨*whileStatement*⟩……這正是那些「多餘」關鍵字（`let`、`do`、`var`）存在的理由。

練習 6：符號表建構

為下列副程式建出完整符號表（名稱、型別、種類、位移）：

```
class Point {
    field int x, y;
    static int count;
    method int distSq(Point other) {
```

```
        var int dx, dy;
        // ...
    }
}
```

解答

類別符號表（field 與 static 分開編位移）：

名稱	型別	種類	位移
x	int	field	0
y	int	field	1
count	int	static	0

副程式符號表（distSq 是**方法**，所以先加 this）：

名稱	型別	種類	位移
this	Point	argument	0
other	Point	argument	1
dx	int	var（local）	0
dy	int	var（local）	1

方法內 x → this 0、other → argument 1、dx → local 0。

練習 7：手工編譯 while

把下列 Jack 編譯成 Hack VM（n 為 local 0）：

```
while (n > 0) {
    let n = n - 1;
}
```

解答

```
label while_start_0
push local 0
push constant 0
gt                      // n > 0
not                     // 條件取反
if-goto while_end_0     // 不成立則離開
push local 0
push constant 1
sub
pop local 0             // n = n - 1
goto while_start_0
label while_end_0
```

模式：條件推上堆疊 → `not` + `if-goto` 結束標籤 → 本體 → `goto` 開始標籤。標籤編號必須全域唯一（巢狀迴圈各有自己的一對）。

練習 8：手工編譯運算式

在練習 6 的 `distSq` 方法內，把 `let dx = x - other[0];` 編譯成 VM 碼。提示：`other[0]` 用 `that` 段。

解答

```
push this 0         // x（field 0，經 this 段）
push argument 1     // other（物件位址）
push constant 0
add                 // other + 0
pop pointer 1       // that 0 -> other[0]
push that 0         // other[0] 的值
sub                 // x - other[0]
pop local 0         // dx =
```

關鍵分工：**欄位走 `this`（pointer 0，方法入口設定、本體不動）、`[]` 下標走 `that`（pointer 1，隨用隨設）**——這正是 2.6 節「that 讓 `[]` 與 field 並用的運算式順利編譯」的意思。

練習 9：方法呼叫的編譯

`p` 是型別 `Point` 的 `local 2`。(a) 把 `do p.distSq(q);`（`q` 為 `local 3`）編譯成 VM 碼。(b) 編譯器如何分辨 `p.distSq(q)` 是**方法**呼叫、而 `Math.sqrt(x)` 是**函式**呼叫？

解答

(a)

```
push local 2          // p 作為隱藏的第一個引數（this）
push local 3          // q
call Point.distSq 2   // 2 個引數（p 算一個！）
pop temp 0            // do 敘述：丟棄返回值
```

(b) 看 `.` 左邊的識別字**是否在符號表中**：`p` 查得到（是變數）⇒ 方法呼叫，推 `p` 為 argument 0、用 `p` 的**型別**（Point）組出完整函式名；`Math` 查不到（是類別名）⇒ 函式／建構子呼叫，不推額外引數。

練習 10：建構子的編譯

類別 `Point` 有 2 個 field（練習 6）。寫出編譯 `constructor Point new(int ax, int ay)` 開頭與結尾所產生的 VM 碼骨架，並解釋每一步。

解答

```
function Point.new 0      // 0 個 local
push constant 2           // 物件大小 = field 數 = 2（查類別符號表）
call Memory.alloc 1       // 在堆積配置 2 個字，返回基底位址
pop pointer 0             // this 段指向新物件
// ...本體：let x = ax; -> push argument 0 / pop this 0
//          let y = ay; -> push argument 1 / pop this 1
push pointer 0            // return this;
return
```

要點：(1) 大小來自**類別符號表**的 field 計數；(2) `pop pointer 0` 建立「`this 0` = 當前物件」的不變式；(3) 建構子**不是**方法——呼叫端不推隱藏引數，`ax` 就是 `argument 0`；(4) 結尾 `push pointer 0` 把新物件位址返回給呼叫者。

練習 11：do 與 void 的堆疊衛生

(a) 為什麼 `do` 敘述編譯後必須 `pop temp 0`？不做會怎樣？(b) void 函式的 `return;` 為什麼要推假值？(c) 呼叫 void 函式後，呼叫端拿到的「返回值」是什麼？

解答

(a) VM 的 `call` **保證**把一個返回值推上堆疊。`do` 敘述不使用它——不彈掉的話，每執行一次 `do`，堆疊就**淨增一個**無主之值。迴圈裡的 `do` 會讓堆疊持續長高，最終溢出堆疊段（256–2047）——「堆疊上的記憶體洩漏」。

(b) 因為 `return` 的實作（第十週）**無條件**把堆疊頂複製到 `ARG` 的位置——堆疊頂必須有東西。慣例是 `push constant 0` 再 `return`。

(c) 那個假值 0。呼叫端（`do` 敘述）隨即 `pop temp 0` 丟棄它——兩端配合，堆疊收支平衡。

練習 12：字串字面值洩漏

(a) 逐步說明 `do Output.printString("Hi");` 在官方編譯法下產生哪些堆積操作。(b) 這段程式在迴圈裡跑 1000 次，堆積淨增多少個 String 物件？(c) 講師的修法中，為什麼「第一個引數是字串字面值」需要特殊處理？

解答

(a) (1) `push constant 2` + `call String.new 1`——在**堆積**配置一個容量 2 的 String；(2) 兩次 `push constant` 字元碼 + `call String.appendChar 2` 填入 `H`、`i`；(3) String 位址留在堆疊頂，作為引數 `call Output.printString 1`；(4) `pop temp 0`。**沒有任何一步釋放那個 String**。

(b) 每圈建一個、無人釋放：**淨增 1000 個**。這就是「Hello, world! 有記憶體洩漏」的具體形式。

(c) 修法要在 `call` **之後**從堆疊上方（SP 之上的殘留區）取回字面值位址來 `dispose`。但 `return` 的實作會把**返回值寫進 argument 0 的位置**（即使函式是 void）——第一個引數的殘留值被覆寫，位址就丟了。解法：第一個引數若是字串字面值，**再推一次**當額外的最後一個引數（`call`

計數 +1；被呼叫者不會用到它，無害），事後改從最後一個引數的位置取回並 `dispose`。

A 附錄：速查表

A.1 Jack 權杖總表

類別	內容
關鍵字（21 個）	`class constructor function method field static; int char boolean void; var let do if else while return; true false null this`
符號（19 個）	`+ - * / & | ~ < > =; { } [ ] ( ) . , ;`
整數字面值	0 . . . 32767
字串字面值	`"..."`（不含換行與雙引號）
識別字	字母／數字／底線，非數字開頭、非關鍵字

A.2 Jack → VM 對照配方

Jack 構造	產生的 VM 碼
整數字面值 n	`push constant n`
`true`	`push constant 0` + `not`（或 `push constant 1` + `neg`）
`false` ／ `null`	`push constant 0`
`this`	`push pointer 0`
var ／ argument ／ static ／ field i	`push local/argument/static/this i`
`a[e]` 讀取	算 `a+e` → `pop pointer 1` → `push that 0`
`x + y`	兩邊各自入棧 → `add`；`*` → `call Math.multiply 2`、`/` → `call Math.divide 2`
`while (c) {s}`	`label S` → c 入棧 → `not` → `if-goto E` → s → `goto S` → `label E`
`if (c) {s1} else {s2}`	c 入棧 → `not` → `if-goto ELSE` → s1 → `goto END` → `label ELSE` → s2 → `label END`
`do f(...);`	引數入棧 → `call` → `pop temp 0`
方法呼叫 `v.m(...)`	`push v`（隱藏引數）→ 其餘引數 → `call Class.m n+1`
方法定義開頭	`push argument 0` → `pop pointer 0`
建構子開頭	`push constant` 欄位數 → `call Memory.alloc 1` → `pop pointer 0`
`return e;`	e 入棧 → `return`；void：`push constant 0` → `return`

A.3 變數種類 → 記憶體段

Jack 宣告	VM 段	生命週期
`var`	`local`	單次副程式呼叫
參數	`argument`	單次副程式呼叫
`static`	`static`	整個程式（同類別共享）
`field`	`this`	物件存活期間（堆積）

A.4 名詞中英對照

英文	中文	英文	中文
expression	運算式	object	物件
statement	敘述	field / static	欄位／靜態變數
operator precedence	運算子優先順序	method	方法
implicit conversion	隱含轉換	constructor	建構子
class	類別	subroutine	副程式
parse tree / AST	解析樹／抽象語法樹	current object	當前物件
recursive descent	遞迴下降	string literal	字串字面值
lookahead	前瞻	memory leak	記憶體洩漏
symbol table	符號表	dangling pointer	懸空指標
scope shadowing	作用域遮蔽	bootstrapping	自舉

A.5 參考資料

- John Lapinskas, *The Jack language ／ Compiling Jack ／ Compiling Jack's classes ／ Summing up and looking forward*（11-1 至 11-4），COMSM1302, University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 9（Jack）、Ch. 10（Compiler I: Syntax Analysis）、Ch. 11（Compiler II: Code Generation）、附錄 5（Hack 字元集）、附錄 6（OS API）.
- nand2tetris Project 10 & 11 （`nand2tetris.org`）——編譯器兩階段開發與漸進式測試程式.