Compiling Jack's classes

John Lapinskas, University of Bristol

How objects work

An instance of a class is called an **object**.

In Jack, every object is a pointer-based array. For any class `Foo`, the code

```
var Foo myFoo;
// Code to initialise myFoo
Output.PrintInt(myFoo);
```

will print the RAM address at which `myFoo` is stored.

How objects work

An instance of a class is called an **object**.

In Jack, every object is a pointer-based array. For any class `Foo`, the code

```
var Foo myFoo;
// Code to initialise myFoo
Output.PrintInt(myFoo);
```

will print the RAM address at which `myFoo` is stored.

If `Foo` has fields `x`, `y` and `z`, defined in that order, then:

- `myFoo.x` is stored at RAM[`myFoo`].
- `myFoo.y` is stored at RAM[$\texttt{myFoo} + 1$], and
- `myFoo.z` is stored at RAM[$\texttt{myFoo} + 2$].

How objects work

An instance of a class is called an **object**.

In Jack, every object is a pointer-based array. For any class `Foo`, the code

```
var Foo myFoo;
// Code to initialise myFoo
Output.PrintInt(myFoo);
```

will print the RAM address at which `myFoo` is stored.

If `Foo` has fields `x`, `y` and `z`, defined in that order, then:

- `myFoo.x` is stored at $\mathrm{RAM}[\texttt{myFoo}]$.
- `myFoo.y` is stored at $\mathrm{RAM}[\texttt{myFoo} + 1]$, and
- `myFoo.z` is stored at $\mathrm{RAM}[\texttt{myFoo} + 2]$.

Given this, it makes sense that `myFoo[i]` evaluates to $\mathrm{RAM}[\texttt{myFoo} + \texttt{i}]$.
So in this example, `myFoo[2]` will evaluate to `myFoo.z`.

How objects work

An instance of a class is called an **object**.

In Jack, every object is a pointer-based array. For any class `Foo`, the code

```
var Foo myFoo;
// Code to initialise myFoo
Output.PrintInt(myFoo);
```

will print the RAM address at which `myFoo` is stored.

If `Foo` has fields `x`, `y` and `z`, defined in that order, then:

- `myFoo.x` is stored at $\text{RAM}[\texttt{myFoo}]$.
- `myFoo.y` is stored at $\text{RAM}[\texttt{myFoo} + 1]$, and
- `myFoo.z` is stored at $\text{RAM}[\texttt{myFoo} + 2]$.

Given this, it makes sense that `myFoo[i]` evaluates to $\text{RAM}[\texttt{myFoo} + \texttt{i}]$.
So in this example, `myFoo[2]` will evaluate to `myFoo.z`.

For simplicity, all object fields in Jack are allocated on the heap.
The pointer `Foo` is stored on the stack like any other `var`.

(This is all very similar to how `structs` work in C, except that they can be allocated on the stack and must deal with different field sizes!)

Desired subroutine behaviour

Recall classes can have functions, methods, and constructors.

Collectively, we call these **subroutines**.[1]

All subroutines `myClass.mySub` of a class `myClass` should do the following:

- On compiling the subroutine call, output Hack VM code which adds the given arguments (which are $\langle\text{expression}\rangle$s) onto the stack, followed by a `call` command to `myClass.mySub`.

---

[1] This is unusual terminology — a subroutine more often refers to a `void` function — but it follows Nisan and Schocken.

Desired subroutine behaviour

Recall classes can have functions, methods, and constructors.

Collectively, we call these **subroutines**.[1]

All subroutines `myClass.mySub` of a class `myClass` should do the following:

- On compiling the subroutine call, output Hack VM code which adds the given arguments (which are $\langle\text{expression}\rangle$s) onto the stack, followed by a `call` command to `myClass.mySub`.
- On compiling the $\langle\text{parameterList}\rangle$ and $\langle\text{varDec}\rangle$s, build a symbol table (see last video). Use it to replace `vars` with numbered `locals` and arguments with numbered `arguments` in the $\langle\text{statements}\rangle$ of the $\langle\text{subroutineBody}\rangle$.

---

[1] This is unusual terminology — a subroutine more often refers to a `void` function — but it follows Nisan and Schocken.

Desired subroutine behaviour

Recall classes can have functions, methods, and constructors.

Collectively, we call these **subroutines**.[1]

All subroutines `myClass.mySub` of a class `myClass` should do the following:

- On compiling the subroutine call, output Hack VM code which adds the given arguments (which are $\langle\text{expression}\rangle$s) onto the stack, followed by a `call` command to `myClass.mySub`.
- On compiling the $\langle\text{parameterList}\rangle$ and $\langle\text{varDec}\rangle$s, build a symbol table (see last video). Use it to replace `var`s with numbered `local`s and arguments with numbered `argument`s in the $\langle\text{statements}\rangle$ of the $\langle\text{subroutineBody}\rangle$.
- On compiling the subroutine return, add the returned $\langle\text{expression}\rangle$ onto the stack if there is one or a dummy value if not, then output a Hack VM `return` command.

---

[1] This is unusual terminology — a subroutine more often refers to a `void` function — but it follows Nisan and Schocken.

Desired subroutine behaviour

Recall classes can have functions, methods, and constructors.

Collectively, we call these **subroutines**.[1]

All subroutines `myClass.mySub` of a class `myClass` should do the following:

- On compiling the subroutine call, output Hack VM code which adds the given arguments (which are $\langle \text{expression} \rangle$s) onto the stack, followed by a `call` command to `myClass.mySub`.
- On compiling the $\langle \text{parameterList} \rangle$ and $\langle \text{varDec} \rangle$s, build a symbol table (see last video). Use it to replace `var`s with numbered `local`s and arguments with numbered `argument`s in the $\langle \text{statements} \rangle$ of the $\langle \text{subroutineBody} \rangle$.
- On compiling the subroutine return, add the returned $\langle \text{expression} \rangle$ onto the stack if there is one or a dummy value if not, then output a Hack VM `return` command.
- In compiling $\langle \text{doStatement} \rangle$s, make sure to e.g. `pop temp 0` after the $\langle \text{subroutineCall} \rangle$ to avoid a "memory leak" onto the stack.

---

[1] This is unusual terminology — a subroutine more often refers to a `void` function — but it follows Nisan and Schocken.

Differences between subroutine types

Functions can disregard their host classes (except for `static` variables). However, both constructors and methods are associated with a **current object** of their class:

- Constructors automatically create their current object on being called, using `Memory.alloc` to allocate a suitably-sized segment on the heap. (The point is to return the current object at the end of the subroutine call.)
- Methods are normally called with the syntax `myVar.myMethod()` rather than `myClass.myMethod()`, and they take `myVar` as their current object.

Differences between subroutine types

Functions can disregard their host classes (except for `static` variables). However, both constructors and methods are associated with a **current object** of their class:

- Constructors automatically create their current object on being called, using `Memory.alloc` to allocate a suitably-sized segment on the heap. (The point is to return the current object at the end of the subroutine call.)
- Methods are normally called with the syntax `myVar.myMethod()` rather than `myClass.myMethod()`, and they take `myVar` as their current object.

Within the bodies of both methods and constructors:

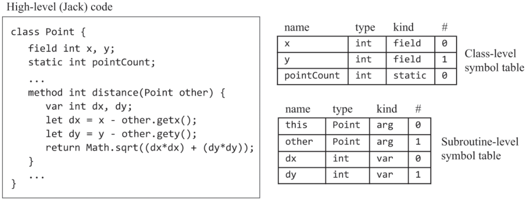
- The `this` keyword evaluates to the current object.
- Any field `x` of the host class evaluates to `this.x`.[2]
- Any method of the host class can be called with the syntax `myMethod()`, and this will be interpreted as `this.myMethod()`.

This is vital for OOP later, but in the context of Jack, it just means you can write e.g. `myToken.write(output)` rather than `write_token(myToken, output)`.

---

[2] Here `this.x` is C syntax, not Jack syntax. Jack doesn't support using `myObject.myField` to refer to an object's field the way C does for structs, as to compile it, you'd need to be able to access information about a class' fields while compiling a different file. That would need a a full pass of semantic analysis across every file in the program — not impossible, but annoying.

The ⟨class⟩ symbol table



High-level (Jack) code

```
class Point {
   field int x, y;
   static int pointCount;
   ...
   method int distance(Point other) {
      var int dx, dy;
      let dx = x - other.getx();
      let dy = y - other.gety();
      return Math.sqrt((dx*dx) + (dy*dy));
   }
   ...
}
```

name	type	kind	#
x	int	field	0
y	int	field	1
pointCount	int	static	0

Class-level symbol table

name	type	kind	#
this	Point	arg	0
other	Point	arg	1
dx	int	var	0
dy	int	var	1

Subroutine-level symbol table

Source: Nisan and Schocken Figure 11.2 (repeat from last video).

Just like with the subroutine class table, we'll only ever need to know where `field` and `static` variables are stored within the code for their class. So on reaching the opening XML tag of a ⟨class⟩, we can:

- Create a new symbol table for the class.
- Add one entry for each variable in each ⟨classVarDec⟩, with separate offsets for `field` and `static` variables. (The `SymbolTable` struct we provide supports this.)
- Use this table while generating code for each ⟨subroutineDec⟩ after the ⟨classVarDec⟩s. (See later.)
- Free the symbol table on reaching the ⟨class⟩ closing tag.

The role of `this`: Implementing methods and constructors

How do we use our class symbol table to compile methods and constructors?

The role of `this`: Implementing methods and constructors

How do we use our class symbol table to compile methods and constructors?

We at last use the `this` segment in Hack VM!

We will ensure that `this 0` is *always* stored at the address pointed to by the current object. If we can do this, then `this[i]` in Jack will always map to `this i` in Hack VM.

The role of `this`: Implementing methods and constructors

How do we use our class symbol table to compile methods and constructors?

We at last use the `this` segment in Hack VM!

We will ensure that `this 0` is *always* stored at the address pointed to by the current object. If we can do this, then `this[i]` in Jack will always map to `this i` in Hack VM.

On compiling a $\langle \text{subroutineCall} \rangle$, we must:

- **For methods only:** Push the current object onto the stack (and add it to the symbol table) as a new first argument before compiling the $\langle \text{expressionList} \rangle$ for the others. Adjust the VM `call` command generated accordingly.
- Distinguish method calls from other subroutine calls by checking to see whether the '`.`' is present, and whether the identifier to its left is a variable.

The role of `this`: Implementing methods and constructors

How do we use our class symbol table to compile methods and constructors?

We at last use the `this` segment in Hack VM!

We will ensure that `this 0` is *always* stored at the address pointed to by the current object. If we can do this, then `this[i]` in Jack will always map to `this i` in Hack VM.

On compiling a $\langle\text{subroutineCall}\rangle$, we must:

- **For methods only:** Push the current object onto the stack (and add it to the symbol table) as a new first argument before compiling the $\langle\text{expressionList}\rangle$ for the others. Adjust the VM `call` command generated accordingly.
- Distinguish method calls from other subroutine calls by checking to see whether the '`.`' is present, and whether the identifier to its left is a variable.

On compiling a $\langle\text{subroutineDec}\rangle$, we must:

- **For methods:** Set `pointer 0` to `argument 0`.
- **For constructors:** Call `Memory.alloc` to allocate a segment for a new object, using the class symbol table to work out how much is needed. Then set `pointer 0` to the base address.
- **For both:** Avoid changing `pointer 0` in the subroutine body!

A summary of subroutine behaviour

	Function	Constructor	Method
Call syntax	`myClass.mySub(a,b)`		`myVar.mySub(a,b)` or `mySub(a,b)`
On call	Normal behaviour		Add `myVar` as argument `0`
On start	Normal behaviour	Set `this` base address to new `myClass` variable	Set `this` base address to `myVar`
In body	Normal behaviour	`myClassVar` is read as `this.myClassVar`, and `myMethod(a)` is read as `this.myMethod(a)`	
On return	Normal behaviour (constructors should always return `this`)		

Compiling ⟨term⟩s

$$
\begin{aligned}
\langle\text{term}\rangle ::= {} & \text{integer literal} \mid \text{string literal} \mid \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'} \mid \text{`}\texttt{null}\text{'} \mid \text{`}\texttt{this}\text{'} \mid \\
& \text{identifier},\ [\text{`}\texttt{[}\text{'},\ \langle\text{expression}\rangle,\ \text{`}\texttt{]}\text{'}] \mid \text{`}\texttt{(}\text{'},\ \langle\text{expression}\rangle,\ \text{`}\texttt{)}\text{'} \mid \\
& ((\text{`}\texttt{-}\text{'} \mid \text{`}\texttt{\textasciitilde}\text{'}),\ \langle\text{term}\rangle) \mid \langle\text{subroutineCall}\rangle;
\end{aligned}
$$

In the `compile_term` function, your goal should be to generate VM code that evaluates the ⟨term⟩'s value and leaves it on top of the stack.

For example, if the ⟨term⟩ is the integer literal 85, then generate `push constant 85`. If the ⟨term⟩ is an ⟨expression⟩ in `()`s, call `compile_expression`.

The hardest cases are when the ⟨term⟩ is an identifier or a string literal. If it's an identifier, look it up in the class and subroutine symbol tables:

Compiling $\langle\text{term}\rangle$s

$$\begin{aligned}\langle\text{term}\rangle ::= {} & \text{integer literal} \mid \text{string literal} \mid \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'} \mid \text{`}\texttt{null}\text{'} \mid \text{`}\texttt{this}\text{'} \mid \\ & \text{identifier},\ [\text{`}\texttt{[}\text{'},\ \langle\text{expression}\rangle,\ \text{`}\texttt{]}\text{'}] \mid \text{`}\texttt{(}\text{'},\ \langle\text{expression}\rangle,\ \text{`}\texttt{)}\text{'} \mid \\ & ((\text{`}\texttt{-}\text{'} \mid \text{`}\texttt{\~{}}\text{'}),\ \langle\text{term}\rangle) \mid \langle\text{subroutineCall}\rangle;\end{aligned}$$

In the `compile_term` function, your goal should be to generate VM code that evaluates the $\langle\text{term}\rangle$'s value and leaves it on top of the stack.

For example, if the $\langle\text{term}\rangle$ is the integer literal 85, then generate `push constant 85`. If the $\langle\text{term}\rangle$ is an $\langle\text{expression}\rangle$ in `()`s, call `compile_expression`.

The hardest cases are when the $\langle\text{term}\rangle$ is an identifier or a string literal. If it's an identifier, look it up in the class and subroutine symbol tables:

- If it's an argument with offset *i*, `push argument i`.

Compiling ⟨term⟩s

⟨term⟩ ::= integer literal | string literal | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;

In the `compile_term` function, your goal should be to generate VM code that evaluates the ⟨term⟩’s value and leaves it on top of the stack.

For example, if the ⟨term⟩ is the integer literal 85, then generate `push constant 85`. If the ⟨term⟩ is an ⟨expression⟩ in `()`s, call `compile_expression`.

The hardest cases are when the ⟨term⟩ is an identifier or a string literal. If it’s an identifier, look it up in the class and subroutine symbol tables:

- If it’s an argument with offset *i*, `push argument i`.
- If it’s a `var` with offset *i*, `push local i`.

Compiling ⟨term⟩s

$$\begin{aligned}\langle\text{term}\rangle ::= {} & \text{integer literal} \mid \text{string literal} \mid \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'} \mid \text{`}\texttt{null}\text{'} \mid \text{`}\texttt{this}\text{'} \mid \\ & \text{identifier},\ [\text{`}\texttt{[}\text{'},\ \langle\text{expression}\rangle,\ \text{`}\texttt{]}\text{'}] \mid \text{`}\texttt{(}\text{'},\ \langle\text{expression}\rangle,\ \text{`}\texttt{)}\text{'} \mid \\ & ((\text{`}\texttt{-}\text{'} \mid \text{`}\texttt{\~{}}\text{'}),\ \langle\text{term}\rangle) \mid \langle\text{subroutineCall}\rangle;\end{aligned}$$

In the `compile_term` function, your goal should be to generate VM code that evaluates the ⟨term⟩'s value and leaves it on top of the stack.

For example, if the ⟨term⟩ is the integer literal 85, then generate `push constant 85`. If the ⟨term⟩ is an ⟨expression⟩ in `()`s, call `compile_expression`.

The hardest cases are when the ⟨term⟩ is an identifier or a string literal. If it's an identifier, look it up in the class and subroutine symbol tables:

- If it's an argument with offset *i*, `push argument i`.
- If it's a `var` with offset *i*, `push local i`.
- If it's a `static` with offset *i*, then `push static i`. Static variables are shared across all objects of a class, so this is valid even in functions.

Compiling ⟨term⟩s

⟨term⟩ ::= integer literal | string literal | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;

In the `compile_term` function, your goal should be to generate VM code that evaluates the ⟨term⟩’s value and leaves it on top of the stack.

For example, if the ⟨term⟩ is the integer literal 85, then generate `push constant 85`. If the ⟨term⟩ is an ⟨expression⟩ in `()`s, call `compile_expression`.

The hardest cases are when the ⟨term⟩ is an identifier or a string literal. If it’s an identifier, look it up in the class and subroutine symbol tables:

- If it’s an argument with offset *i*, `push argument i`.
- If it’s a `var` with offset *i*, `push local i`.
- If it’s a `static` with offset *i*, then `push static i`. Static variables are shared across all objects of a class, so this is valid even in functions.
- If it’s a `field` with offset *i*, then for this to be valid Jack code, we must be in a method or constructor. In that case, it belongs to the current object, which is always stored in `this` at `pointer 0`. Remember that objects are stored as arrays in RAM, with the *i*’th field in position *i* — so `push this i` will do the job.

Compiling ⟨term⟩s

⟨term⟩ ::= integer literal | string literal | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;

In the `compile_term` function, your goal should be to generate VM code that evaluates the ⟨term⟩’s value and leaves it on top of the stack.

For example, if the ⟨term⟩ is the integer literal 85, then generate `push constant 85`. If the ⟨term⟩ is an ⟨expression⟩ in `()`s, call `compile_expression`.

The hardest cases are when the ⟨term⟩ is an identifier or a string literal. If it’s an identifier, look it up in the class and subroutine symbol tables:

- If it’s an argument with offset *i*, `push argument i`.
- If it’s a `var` with offset *i*, `push local i`.
- If it’s a `static` with offset *i*, then `push static i`. Static variables are shared across all objects of a class, so this is valid even in functions.
- If it’s a `field` with offset *i*, then for this to be valid Jack code, we must be in a method or constructor. In that case, it belongs to the current object, which is always stored in `this` at `pointer 0`. Remember that objects are stored as arrays in RAM, with the *i*’th field in position *i* — so `push this i` will do the job.

If it appears in both the class and subroutine tables, prioritise the subroutine table.

String literals: The official way

If the $\langle \text{term} \rangle$ is a string literal, the official nand2tetris compilation method is:

- Create a new `String` of the right maximum length with `String.new`.
- Initialise the string to match the literal with calls to `String.appendChar`.
  - Converting from C `chars` to Hack VM integers to pass to `String.appendChar` is easy, since the Hack character set aligns with ASCII (see Nisan and Schocken Appendix 5) — so you can just cast to int.
- Push the new `String`'s address onto the stack.

That solves the problem, right?

String literals: The official way

If the $\langle \text{term} \rangle$ is a string literal, the official nand2tetris compilation method is:

- Create a new `String` of the right maximum length with `String.new`.
- Initialise the string to match the literal with calls to `String.appendChar`.
  - Converting from C `chars` to Hack VM integers to pass to `String.appendChar` is easy, since the Hack character set aligns with ASCII (see Nisan and Schocken Appendix 5) — so you can just cast to int.
- Push the new `String`'s address onto the stack.

That solves the problem, right?

What happens when someone calls `Output.printString("Uh-oh!")` in a loop?

String literals: The official way

If the $\langle\text{term}\rangle$ is a string literal, the official nand2tetris compilation method is:

- Create a new `String` of the right maximum length with `String.new`.
- Initialise the string to match the literal with calls to `String.appendChar`.
    - Converting from C `char`s to Hack VM integers to pass to `String.appendChar` is easy, since the Hack character set aligns with ASCII (see Nisan and Schocken Appendix 5) — so you can just cast to int.
- Push the new `String`'s address onto the stack.

That solves the problem, right?

What happens when someone calls `Output.printString("Uh-oh!")` in a loop? The code for the $\langle\text{expression}\rangle$ `"Uh-oh!"` gets run every single time...

String literals: The official way

If the $\langle\text{term}\rangle$ is a string literal, the official nand2tetris compilation method is:

- Create a new `String` of the right maximum length with `String.new`.
- Initialise the string to match the literal with calls to `String.appendChar`.
  - Converting from C `char`s to Hack VM integers to pass to `String.appendChar` is easy, since the Hack character set aligns with ASCII (see Nisan and Schocken Appendix 5) — so you can just cast to int.
- Push the new `String`'s address onto the stack.

That solves the problem, right?

What happens when someone calls `Output.printString("Uh-oh!")` in a loop? The code for the $\langle\text{expression}\rangle$ `"Uh-oh!"` gets run every single time... which creates a new `String` each time...

String literals: The official way

If the ⟨term⟩ is a string literal, the official nand2tetris compilation method is:

- Create a new `String` of the right maximum length with `String.new`.
- Initialise the string to match the literal with calls to `String.appendChar`.
    - Converting from C `char`s to Hack VM integers to pass to `String.appendChar` is easy, since the Hack character set aligns with ASCII (see Nisan and Schocken Appendix 5) — so you can just cast to int.
- Push the new `String`'s address onto the stack.

That solves the problem, right?

What happens when someone calls `Output.printString("Uh-oh!")` in a loop? The code for the ⟨expression⟩ `"Uh-oh!"` gets run every single time... which creates a new `String` each time... which is allocated on the heap before being passed to `Output.printString`...

String literals: The official way

If the $\langle\text{term}\rangle$ is a string literal, the official nand2tetris compilation method is:

- Create a new `String` of the right maximum length with `String.new`.
- Initialise the string to match the literal with calls to `String.appendChar`.
    - Converting from C `char`s to Hack VM integers to pass to `String.appendChar` is easy, since the Hack character set aligns with ASCII (see Nisan and Schocken Appendix 5) — so you can just cast to int.
- Push the new `String`'s address onto the stack.

That solves the problem, right?

What happens when someone calls `Output.printString("Uh-oh!")` in a loop? The code for the $\langle\text{expression}\rangle$ `"Uh-oh!"` gets run every single time... which creates a new `String` each time... which is allocated on the heap before being passed to `Output.printString`... **and then is never freed**.

"Hello, world!" in Jack has a memory leak!

Should we fix this?

When we talk about "fixing string literals in Jack", we should be clear what we mean. In one sense, they aren't broken. The language is behaving as specified. If we free string literals automatically to prevent this sort of memory leak, we will break existing Jack code (and the test scripts).

Source: xkcd 1172. Alt text: "There are probably children out there holding down spacebar to stay warm in the winter! YOUR UPDATE MURDERS CHILDREN."

Should we fix this?

When we talk about “fixing string literals in Jack”, we should be clear what we mean. In one sense, they aren’t broken. The language is behaving as specified. If we free string literals automatically to prevent this sort of memory leak, we will break existing Jack code (and the test scripts).

More seriously, we can see there’s no easy answer by looking at C.

`char myString[] = "Hello, world!";` sets `myString` to a copy of the string “Hello, world!” stored on the stack. It disappears on function return and can be modified as normal until then.

Should we fix this?

When we talk about "fixing string literals in Jack", we should be clear what we mean. In one sense, they aren't broken. The language is behaving as specified. If we free string literals automatically to prevent this sort of memory leak, we will break existing Jack code (and the test scripts).

More seriously, we can see there's no easy answer by looking at C.

`char myString[] = "Hello, world!";` sets `myString` to a copy of the string "Hello, world!" stored on the stack. It disappears on function return and can be modified as normal until then.

So if you `return myString;` from a function that returns `char *`, it will be left as a dangling pointer. Hmm.

Should we fix this?

When we talk about "fixing string literals in Jack", we should be clear what we mean. In one sense, they aren't broken. The language is behaving as specified. If we free string literals automatically to prevent this sort of memory leak, we will break existing Jack code (and the test scripts).

More seriously, we can see there's no easy answer by looking at C.

`char myString[] = "Hello, world!";` sets `myString` to a copy of the string "Hello, world!" stored on the stack. It disappears on function return and can be modified as normal until then.

So if you `return myString;` from a function that returns `char *`, it will be left as a dangling pointer. Hmm.

Meanwhile `char *myString = "Hello, world!";` creates, at the start of program execution, a **single static copy** of the string "Hello, world!" in memory. `myString` will then be initialised as a pointer to this copy.

Should we fix this?

When we talk about "fixing string literals in Jack", we should be clear what we mean. In one sense, they aren't broken. The language is behaving as specified. If we free string literals automatically to prevent this sort of memory leak, we will break existing Jack code (and the test scripts).

More seriously, we can see there's no easy answer by looking at C.

`char myString[] = "Hello, world!";` sets `myString` to a copy of the string "Hello, world!" stored on the stack. It disappears on function return and can be modified as normal until then.

So if you `return myString;` from a function that returns `char *`, it will be left as a dangling pointer. Hmm.

Meanwhile `char *myString = "Hello, world!";` creates, at the start of program execution, a **single static copy** of the string "Hello, world!" in memory. `myString` will then be initialised as a pointer to this copy.

So if you run `myString[0] = 'J';`, you get a segfault. Oh *dear*.

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | ~~**string literal**~~ | ‘true’ | ‘false’ | ‘null’ | ‘this’ |
identifier, [‘[’, ⟨expression⟩, ‘]’] | ‘(’, ⟨expression⟩, ‘)’ |
((‘-’ | ‘~’), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= ‘let’, identifier, [‘[’, ⟨expression⟩, ‘]’], ‘=’, ⟨expression⟩, ‘;’ | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {‘,’, (⟨expression⟩ | **string literal**)}];

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | ~~string literal~~ | '`true`' | '`false`' | '`null`' | '`this`' |
identifier, ['`[`', ⟨expression⟩, '`]`'] | '`(`', ⟨expression⟩, '`)`' |
(('`-`' | '`~`'), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= '`let`', identifier, ['`[`', ⟨expression⟩, '`]`'], '`=`', ⟨expression⟩, '`;`' | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {'`,`', (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | ~~**string literal**~~ | '`true`' | '`false`' | '`null`' | '`this`' |
identifier, ['`[`', ⟨expression⟩, '`]`'] | '(', ⟨expression⟩, ')' |
(('`-`' | '`~`'), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= '`let`', identifier, ['`[`', ⟨expression⟩, '`]`'], '`=`', ⟨expression⟩, '`;`' | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {'`,`', (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

In compiling an ⟨expressionList⟩ as part of a ⟨subroutineCall⟩, though, we really should automatically free the string.

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | **~~string literal~~** | '`true`' | '`false`' | '`null`' | '`this`' |
identifier, ['`[`', ⟨expression⟩, '`]`'] | '`(`', ⟨expression⟩, '`)`' |
(('`-`' | '`~`'), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= '`let`', identifier, ['`[`', ⟨expression⟩, '`]`'], '`=`', ⟨expression⟩, '`;`' | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {'`,`', (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

In compiling an ⟨expressionList⟩ as part of a ⟨subroutineCall⟩, though, we really should automatically free the string.

- Have `compile_expression_list` pass a list of string literal arguments back to `compile_subroutine_call`.

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | ~~**string literal**~~ | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= ‘`let`’, identifier, [‘`[`’, ⟨expression⟩, ‘`]`’], ‘`=`’, ⟨expression⟩, ‘`;`’ | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {‘`,`’, (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

In compiling an ⟨expressionList⟩ as part of a ⟨subroutineCall⟩, though, we really should automatically free the string.

- Have `compile_expression_list` pass a list of string literal arguments back to `compile_subroutine_call`.
- After generating the VM `call` command, the function arguments will still be left on the stack (above the current stack pointer).

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | **~~string literal~~** | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= ‘`let`’, identifier, [‘`[`’, ⟨expression⟩, ‘`]`’], ‘`=`’, ⟨expression⟩, ‘`;`’ | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {‘`,`’, (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

In compiling an ⟨expressionList⟩ as part of a ⟨subroutineCall⟩, though, we really should automatically free the string.

- Have `compile_expression_list` pass a list of string literal arguments back to `compile_subroutine_call`.
- After generating the VM `call` command, the function arguments will still be left on the stack (above the current stack pointer).
- Retrieve all the string literals and call `String.dispose` on them.

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | ~~**string literal**~~ | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= ‘`let`’, identifier, [‘`[`’, ⟨expression⟩, ‘`]`’], ‘`=`’, ⟨expression⟩, ‘`;`’ | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {‘`,`’, (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

In compiling an ⟨expressionList⟩ as part of a ⟨subroutineCall⟩, though, we really should automatically free the string.

- Have `compile_expression_list` pass a list of string literal arguments back to `compile_subroutine_call`.
- After generating the VM `call` command, the function arguments will still be left on the stack (above the current stack pointer).
- Retrieve all the string literals and call `String.dispose` on them.

Problem: The first function argument will be overwritten by the return value at the end of the function call. (Even if the function is `void`!)

String literals: The unofficial version

All that said, here's the Hack I came up with. First, modify the Jack grammar:

⟨term⟩ ::= integer literal | ~~**string literal**~~ | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ |
identifier, [‘`[`’, ⟨expression⟩, ‘`]`’] | ‘`(`’, ⟨expression⟩, ‘`)`’ |
((‘`-`’ | ‘`~`’), ⟨term⟩) | ⟨subroutineCall⟩;
⟨letStatement⟩ ::= ‘`let`’, identifier, [‘`[`’, ⟨expression⟩, ‘`]`’], ‘`=`’, ⟨expression⟩, ‘`;`’ | **string literal**;
⟨expressionList⟩ ::= [(⟨expression⟩ | **string literal**), {‘`,`’, (⟨expression⟩ | **string literal**)}];

In compiling a ⟨letStatement⟩, use the official way. If someone is explicitly creating a pointer to a string literal, it's up to them to call `String.dispose` to free it later.

In compiling an ⟨expressionList⟩ as part of a ⟨subroutineCall⟩, though, we really should automatically free the string.

- Have `compile_expression_list` pass a list of string literal arguments back to `compile_subroutine_call`.
- If the first argument is a string literal, push it onto the stack again as another argument and increase the argument count of the `call` command appropriately.
- After generating the VM `call` command, the function arguments will still be left on the stack (above the current stack pointer).
- Retrieve all the string literals and call `String.dispose` on them. If the first argument is a string literal, call `String.dispose` on the last argument instead.