The Data Stack

Data in and out of your programs

One of the key ideas about Smojo is that it implicitly uses a **data stack** to direct input and output into and out of words.

WARNING: If you have used other programming languages, this concept will probably be unfamiliar to you. But is important to understand if you want to program Smojo effectively.

What is a Stack?

A stack is something that can store information or data - numbers, text or other kinds of information.

Stacks are an important Data Structure used in writing software. In Smojo, there is one data stack used internally to shuttle information in and out of words

You can just do two things to a stack:

- You can **push** an item onto it.
- You can take an item off of it. This is called **pop**ping an item from the stack. Popping an item removes it from the stack.

Last in First Out

There is a crucial Rule of Stack that governs which items are popped off the stack:

- The **last item** pushed onto the stack is the **first item** that may be popped from it.
- This is called the "last-in-first-out" rule or LIFO for short.
- You **cannot select** which item to pop from the stack. That choice is rigidly made using the LIFO rule.

Stack Simulation

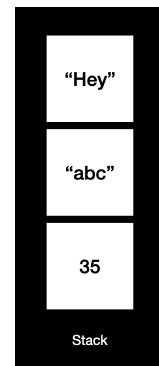

Figure 1: A pictorial representation of our stack. Items are always pushed onto the top of the stack.

Let's see how this works in practice. Let's say you pushed two items in sequence, first the number `35` then `"abc"` then the text `"Hey"`. (See Figure 1)

If you popped the stack now, what item do you expect to get?

Here's how to reason about it:

a. The last item pushed onto the stack is the text `"Hey"`

b. Therefore, using the LIFO rule, it should be the first item popped out of the stack (Figure 2)

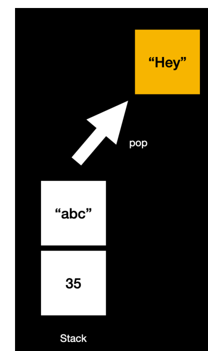

Figure 2: An item is popped from the stack the first time.

c. If it is popped off the only remaining items are `35` and `"abc"`

d. If the stack is popped again, the item `"abc"` is obtained. The stack now contains just the number `35`. (Figure 3)

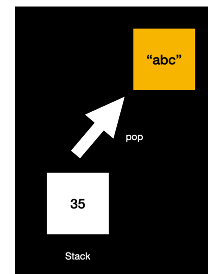

Figure 3: An item is popped from the stack the second time.

e. If the stack is popped a third time, then the sole item, `35` is obtained. At this point, the stack is empty.

f. Popping the stack a forth time will result in an error, since they stack is already empty.

Go through these steps in your mind until it is completely clear.

Smojo's Data Stack

Now that you understand what a stack is, let's see how it applies specifically to Smojo. Key concept:

> "Every data item in Smojo is either being implicitly pushed on to a single data stack or implicitly popped from it."

Let's unpack what this means. Take the following program:

```
1 2 3 + .
```

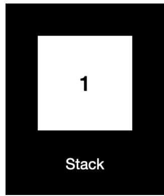

Figure 4: The action of the literal **1** is to push the number 1 onto the data stack.

"A literal is something that pushes its value onto the data stack"

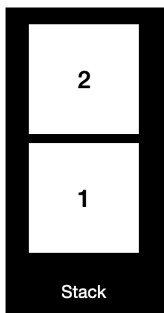

Figure 5: The action of the literal **2** is to push the number 2 onto the data stack.

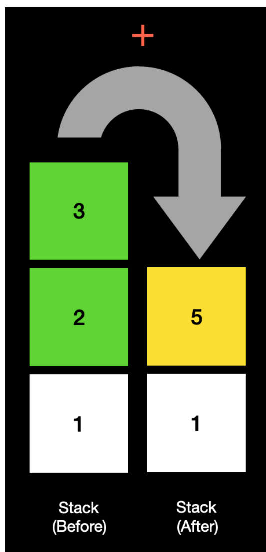

Figure 6: The action of the word **+** is to pop the top two items, add them and push the result onto the data stack.

At first, the data stack is empty.

Recall that Smojo reads your program from left to right. The first thing in this program is the literal **1** , whose action is to push the number **1** onto the stack. So the stack at this point contains just one item, the number **1** (Figure 4).

Note that there is a real difference between the literal **1** and the number **1**:

- The literal **1** has an action associated with it - it pushes the number **1** onto the stack.
- The number **1** represents a numerical value. It does not have an action attached to it.

The second item Smojo sees is the literal **2**. This pushes the number **2** onto the data stack. This results in two items on the data stack - **1** at the bottom and **2** on top (Figure 5).

A similar thing happens for the next literal, **3**. Now the stack is **1 2 3**, with **1** on the bottom and **3** on the top.

The next item is **+**, which is a word, whose action is to pop two items from the stack, add them then push the result onto the data stack.

These top two items are the numbers **2** and **3**, so they are popped by **+**, which also adds together to give the number **5** which is promptly pushed onto the data stack (Figure 6).

The stack now contains just two items: **1 5**

Lastly, the period **.** word pops the data stack and prints the result. Popping the stack **1 5** yields the number **5** (which is displayed to you) and leaves behind the number **1** on the stack.

The program ends with a single item, **1** on the data stack.

Quiz 2.0

2.0.0: Analyse the program:

```
1 2 3 + + .
```

What does it do? Does it leave anything on the stack?

2.0.1: Is the program:

```
1 2 + 3 + .
```

different from the one in 2.0.0? Explain your answer.

Words and the Data Stack

Let's analyse a longer program:

```
1 : greet ( "s" – )
2     "Hello" . . cr
3 ;
4 "Joel" greet
```

The line numbers `1` to `4` are shown on the left for convenience, but they are not part of the program.

The code in purple, ie `( "s" – )` is a **stack comment**. It's a comment, meaning that it is actually ignored completely by Smojo. However, it is useful to programmers: this one tells us that the input into `greet` is a single string/text, and that it outputs nothing.

The other thing that you need to see is that this program is written in two **modes**. The section:

```
: greet ( "s" – )
    "Hello" . . cr
;
```

is mainly in **compilation mode**. This means that although Smojo reads from left to right, words like `.` and `cr` on the second line **aren't** immediately executed. Instead, a new word is being defined. In this case the word `greet`.

The second section is:

```
"Joel" greet
```

which is in **interpretation mode**. This part is indeed executed immediately as it is being read.

For now, simply accept that anything between `:` and `;` is in compilation mode — ie, Smojo is defining a new word.

NOTE: The situation is actually a lot more complex. As a programmer you are given fine control over these modes, which is important in metaprogramming. But more of this in later chapters.

You may interleave compilation mode code with interpretation mode code in your programs. But it's usually a good idea to make all your word definitions first before you use interpretation mode. But there are many exceptions to this rule.

With that out of the way, let's analyse the program!

During the definition of `greet` the data stack is never used, because everything is compiled (ie, added into) the definition of `greet`. So the data stack remains empty throughout this compilation mode stage.

`"Joel"` is a string literal which pushes the string `"Joel"` onto the data stack.

`greet` executes the code compiled within it, ie:

```
"Hello" . . cr
```

`"Hello"` pushes the string `"Hello"` onto the stack. The stack now contains: `"Joel" "Hello"`

The first . pops **“Hello”** and prints it. The second . pops **“Joel”** and prints it (Figure 7). The cr prints the new line.

The program then ends, with nothing on the stack.

The key insight here is that the usage of greet (on line 4)

```
“Joel” greet
```

is equivalent to:

```
“Joel” “Hello” . . cr
```

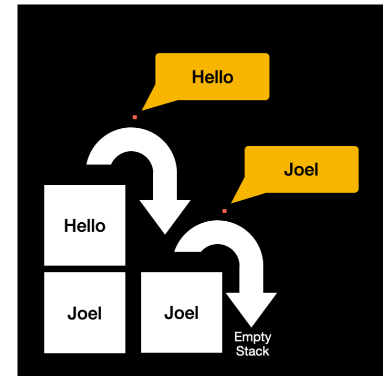

Figure 7: The action of the two periods . .

Quiz 2.1

2.1.0: Write a word holiday (“s” –) that when used:

```
“Deepavali” holiday
```

says **Happy Deepavali ok**

Test out your word for other holidays. Analyse the data stack when “Christmas” holiday is called.

***2.1.1:** Amend the holiday word so that it prints out the number of times it is called. Eg,

```
“Deepavali” holiday
“Christmas” holiday
“Hari Raya” holiday
“Chinese New Year” holiday
```

should print out:

```
Happy Deepavali 1
Happy Christmas 2
Happy Hari Raya 3
Happy Chinese New Year 4
```

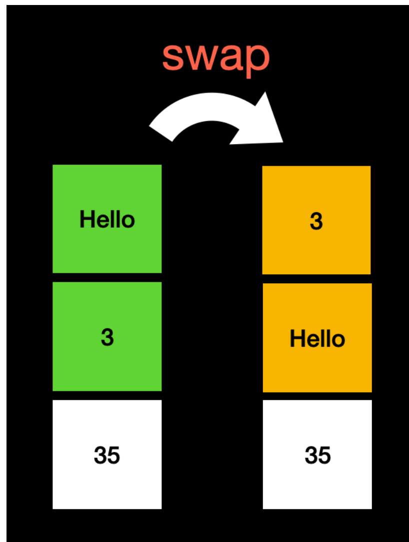

Figure 8: The action of `swap` on the data stack. Only the top two items (in this case, `Hello` and 3) are switched; the rest of the stack (in this case the number `35`) are left as they are.

Rearranging the Data Stack

In many cases you want to re-arrange items on the stack. For example, if you **already have** 3 and 5 on the data stack (eg, as output from a word), the `-` word always subtracts its second argument from the first. That is, `3 5 -` is equivalent to `3 - 5`.

How would you create a version of `-` (let's call it `-*`) that subtracts the second argument from the first? Ie, `3 5 -*` is the same as `5 - 3`.

There are two ways to so this:

We could use the stack word `swap` to switch the places of the top two items:

```
: -* ( n n – n )
     swap -
;
```

Figure 8 shows the action of `swap` on the data stack.

The second method - which is slower and more verbose - is to use **locals**. These are temporary names we give to items on the stack. For example, let's give the name `1st` to the first argument and `2nd` to the second one. Here's how we would write `-*` using locals:

```
: -* ( n n – n ) { 1st 2nd }
   2nd 1st -
;
```

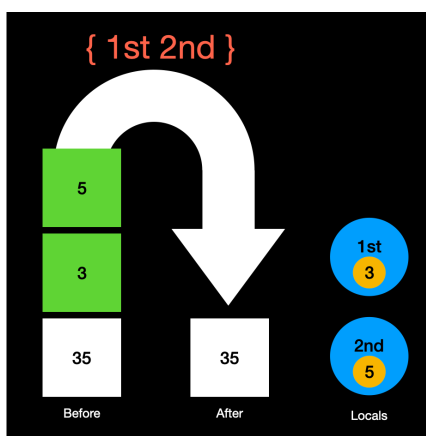

Figure 9: The action of using locals is to remove items from the data stack and to assign these items temporary name. In this example, `1st = 3` and `2nd = 5`, which can be used throughout this word.

- The word `{` begins a locals definition, and the definition is stopped when the } is met. } itself is not a word, it is simply a signal to `{` to stop the definition.

- The names `1st` and `2nd` are assigned to items on the data stack in the order they are met. The actual items on the stack are **removed** once the definitions are done. Figure 9 illustrates this.

- You can think of `1st` or `2nd` as temporary words whose action is to push the value they represent onto the data stack.

The important thing to note about locals is that they remove items from the data stack and give these items

you've defined. These names are temporary in the sense that they may only be used in the word within which the local is defined.

Locals can only be defined and used within word (compilation mode), while stack words can be used in either compilation or interpretation mode.

- Use stack words for simple rearrangement. The example above is a good application. **Do not** use locals for such cases as they will make your code harder to understand. And slower.

- Use locals only for complex rearrangement. Even in this case, there is usually the need to mix using stack words and locals.

As with all things in life, plenty of practice and reading good code will help you get a feeling of when to use them.

Common Stack Words

Some common stack words:

- `swap` **switches** the top two items.
- `drop` **discards** the top item on the data stack.
- `dup` **duplicates** the top item on the data stack.

Quiz 2.2

2.2.0: Without running it, what is the output of the following program?

```
1 2 dup + -
```

2.2.1: Write a word `nip` that drops the second item from the top. Eg, if `nip` is applied to a stack `1 2 3 4`, we want the result to be `1 2 4` , ie the second item has been discarded. Hint: You only need to use the common stack words.

2.2.2: Without using locals, is it possible to write a word `over` that takes the second item from the top, duplicates it and places it at the top of the stack? ie, `1 2 3 over` will become `1 2 3 2`?

The Spare Stack

Quiz 2.2.2 shows us that to make the stack words more complete, we need a place to temporarily store data outside the data stack.

That place is the **spare stack**.

As the name implies, it is also a stack. You can push items from the data stack into it using **>R** (called PUSH-R) and pop items off it back into the data stack using **R>** (called POP-R).

Here is a sequence of actions that illustrate the action of these words:

```
1 2 3 >R \ 1 2 on data stack
         \ 3 on spare stack
+        \ adds 1 and 2 on the data stack
         \ 3 on data stack
R>       \ data stack is 3 3
         \ spare stack is empty
-        \ data stack contains number 0
```

In addition to these words, there is also **R@** which is called PEEK-R, that simply pushes a copy of the top of spare stack onto the data stack.

Quiz 2.3

2.3.0: Without running this code, can you say what the result will be? `1 2 >R >R R@ .`

2.3.1: Create a word `rdrop` that removes the top of the spare stack.

2.3.2: Write the over word as described in Quiz 2.2.2.

2.3.3: Write a word tuck that inserts the top item into 3rd place. Eg: `1 2 3 tuck` becomes `1 3 2 3`

NOTE: `rdrop`, `over` and `tuck` are also common stack words.

The stack is an integral part of programming Smojo, so you need to be really comfortable working with it.

If you program in other languages, you probably will be familiar with the concept of locals. That's good! But you will also likely use locals reflexively. That's bad.

Learn to use stack words well.

Stack Viewing and Debugging

Most of the problems in Smojo go on in the data stack:

- You might expect and item to be on the data stack when it's not. This will give the dreaded "null" error or "Nullpointer Exception" error.
- The places of the items might be different from what you expect. This may give you bugs or more likely a cryptic "Class Cast Exception" error.

Smojo has two simple but powerful words to help you debug these situations:

.S prints the contents of the data stack without altering it. For example,

```
1 2 3 .S
```

will display

```
<3> 1 2 3 ok
```

BP is similar, but prints information of the word that it's in. It can be used in both compilation and interpretation mode and I would recommend you use this in your code for all basic debugging. Here's an example using our **-***:

```
: -* ( n n — n )
    swap bp - bp
;
```

I've used a different colour for **BP** just to make the code easier to read, and to emphasise that **BP** has **no effect** on your underlying code.

Try running this program:

```
1 2 -* .
```

with the this definition of `-*` and you should see:

```
--breakpoint: -*[1] --
<2> 2 1

--breakpoint: -*[2] --
<1> 1
1 ok
```

Note that:

- The name of the word in which **BP** is called (in this case, `-*`) is displayed.
- The **index** of the breakpoint within the `-*` word is also displayed in square brackets. In this example, there are two BPs in -* and their index is clearly shown.
- **BP** displays the contents of the data stack (ie, the same as `.S`).

Quiz 2.4

2.4.0: The word `.R` displays the contents of the Spare stack. Redefine **BP** so that it also displays the contents of the Spare stack. Test your code with the example using `-*`.

2.4.1: Can you find the bug in the following code? Use **BP** to assist you:

```
: hello ( -- "s" )
   "Hello" .
;

: greet ( "s" -- )
   hello . . cr
;

"Maurice" greet
```

Pro Tip

Since Smojo is executed top-to-bottom, left-to-right, you can use this to your advantage to narrow down the location of most errors.

If your **BP** displays **before** an error is shown, then you know that the error occurs at a point in code **after** that **BP**.

Similarly if an error is displayed before your **BP**, then the buggy code occurs before the **BP**.

You can therefore strategically place BPs to narrow down the location of an error.

Learning Points

- You understand what a stack is and how Smojo used a single stack called the “data stack” to passes data between words.
- You know how to manipulate items on the data stack using stack words and locals.
- You know how to use BP to debug your programs.

Answers to Quizzes

Quiz 2.0

2.0.0: It prints 6 and leaves nothing on the data stack.

2.0.1: Exactly the same as **2.0.0** - it prints 6 and leaves nothing on the data stack. Although they are equivalent, this form is preferable to **2.0.0** as the actions of **+** can be clearly understood.

Quiz 2.1

2.1.0:

```
: holiday ( “s” — )
   “Happy” . . cr
;
```

2.1.1:

```
: holiday ( n “s” — n ) { n s }
  “Happy” . s .
  n . cr
  n 1 +
;
1 \ Start index
“Deepavali” holiday
“Christmas” holiday
“Hari Raya” holiday
“Chinese New Year” holiday
```

Quiz 2.2

2.2.0: **-3** on the data stack.

2.2.1:

```
: nip ( a b – b )
   swap drop
;
```

2.2.2: Not possible since we need to save the intermediate result somewhere.

Quiz 2.3

2.3.0: `1 ok`

2.3.1:

```
: rdrop ( – )
  R> drop
;
```

2.3.2:

```
: over ( a b – a b a )
   >R dup R> swap
;
```

2.3.3:

```
: tuck ( a b – b a b )
   swap over
;
```

Quiz 2.4

2.4.0:

```
: BP ( – )
   BP .R
;
```

2.4.1: Can you find the bug in the following code? Use **BP** to assist you:

```
: hello ( -- "s" )
   bp
   "Hello" .
   bp
;

: greet ( "s" -- )
   bp
   hello . . cr
   bp
;

"Maurice" greet
```

When executed, shows the following:

```
--breakpoint: GREET[1] --
<1> Maurice

--breakpoint: HELLO[1] --
<1> Maurice
Hello
--breakpoint: HELLO[2] --
<1> Maurice
Maurice ERROR: null
... in line 18
java.lang.ArrayIndexOutOfBoundsException
ok
```

This output clearly tells us the issue:

- The second `BP` in `greet` doesn't display, while all others do, meaning that the error occurs **after** `hello` is called but **before** the last `BP`, ie, in: `. . cr`

- `CR` is unlikely the problem (since it just prints out a new line without affecting the data stack), so the issue is with the double periods. They expect 2 items on the data stack. But the `BP` after `hello` is run shows only one item (**`Maurice`**)

- The stack comment on `hello` is `( – "s" )` which means it needs to output a single string. But the `BP`s when `hello` starts and ends shows no change in the stack size. We should expect the data stack size to increase from 1 to 2 after `hello` is run, but it actually remains the same. This is the bug.

- We then notice that `hello` has a period `.` that prints `Hello`. Removing this period solves the bug.

Appendix: Stack Words

Word	Action
`drop`	discards an item from the data stack. Eg, `1 2 3 drop` will yield `1 2` on the data stack.
`dup`	Duplicates the top item on the data stack. Eg, `1 2 3 dup` will yield `1 2 3 3`
`swap`	swaps the top two items on the data stack. Eg, `1 2 3 swap` yields `1 3 2`
`over`	`1 2 3 over` becomes `1 2 3 2`
`tuck`	`1 2 3 tuck` becomes `1 3 2 3`
`nip`	`1 2 3 nip` becomes `1 3`
`>R R> R@`	Pushes, pops and peeks items into the spare stack.
`.S`	Displays the contents of the data stack without altering it.
`BP`	Displays the contents of the data stack along with information on which word calls it. Very useful in debugging.
`.R`	Prints contents of the Spare stack.