

TWO: FILLING IN BLANKS

We've seen how you can use tasks as templates that fill in parts of themselves using other tasks. But what if you want them to fill in the blanks using text you specify yourself? You can do that by adding a **parameter** to your task. Parameters are extra information you specify after the name of the task when you run it or write a method for it. For example,

```
[randomly]
Greet ?who: Hello, ?who.
Greet ?who: Hey, ?who.
Greet ?who: Good morning, ?who.
```

This says that Greet takes one piece of extra information, namely who is being greeted. When you run Greet, you specify who should be greeted. So instead of just typing "Greet" to run Step, you'd type something like:

```
Greet "Mr. Daniels"
```

And this would generate text such as "Hello, Mr. Daniels." or "Good Morning, Mr. Daniels." The quotation marks around "Mr. Daniels" are there so that if Greet takes more than one parameter, it can tell where the text of one ends and where the text of another begins. They're not necessary if the text you specify is in lower case and doesn't have any spaces in it. So you can just say:

```
Greet dawg
```

Which is a little less typing than:

```
Greet "dawg"
```

But they both mean the same thing. Both can generate text such as "Hey, dawg."

VARIABLES

We've carefully stepped around the question of what "?who" means in the methods above. You may have guessed that it's a placeholder that can be filled in when Greet is called, and then used in the text template to fill in blanks. These placeholders are called **variables**. In particular, they're called "local variables" because if two different methods both have a variable named ?who, they're treated as different variables so they don't interfere with one another (if you don't get why that might be an issue, don't worry about it).

FOR PROGRAMMERS

Since some of you will want to know, here are the lexical conventions of Step:

- Parameters are separated by spaces, not commas.
- A parameter that looks like a number is treated as a number: -1, 8.6, etc.
- A parameter that starts with a ? is a variable local to that method. For example, ?who, ?x, ?myVariable, or ?my_variable.
- A parameter that starts with a capital letter is a global variable. Tasks are stored in global variables, but global variables don't have to have tasks as their values.
- Quoted strings in parameter lists are treated as single string, as with other languages.
- Any other parameter not covered by the above rules is just a string. So in the "Greet dawg" example, it's passed a string, "dawg" as an argument, whereas "Greet x y z" would be passed three arguments, the strings "x", "y", and "z".

To print the text in a variable, just list it on the right-hand side of the colon in a method:

```
Greet ?who: Good evening, ?who.
```

Variables can also be used passed on as parameters to other tasks. For example:

```
MissionBreifing ?who ?mission:  
[Greet ?who] Your mission, should you decide to  
accept it, is to ?mission.  
[end]
```

This task takes two parameters, who we're briefing (?who), and what their mission is (?mission). If we run:

```
MissionBriefing "Mr. Phelps" "save the world from  
fascism"
```

It will start by calling Greet with ?who as its parameter. But ?who is just a placeholder for "Mr. Phelps". So the system prints "Good evening, Mr. Phelps." Then it prints the "your mission" stuff and finally the mission and a period. So we get:

```
Good evening, Mr. Phelps. Your mission, should you decide to  
accept it, is to save the world from fascism.
```

Notice, by the way, that we named this task MissionBriefing and not "Mission briefing". The reason for that is that if we put a space in the middle of the task name, Step would think it was named "Mission" and "briefing" was a parameter to it. So variables and task names can't have spaces in them. Parameters can, but only if you use quotation marks to make clear that you mean the words in quotation marks as a single parameter.

MATCHING PATTERNS IN PARAMETERS

Suppose we want to have a task that prints out statements of the form "?a gave ?b the ?c". We could do this with a task with three parameters:

```
Give ?giver ?receiver ?item:  
?giver gave ?receiver?item.  
[end]
```

And if you say:

```
Give "Mary" "Jill" "a red rose"
```

And it will say:

FOR CS MAJORS

Local variables get assigned values through a matching process called "unification."

When you call a task T, the system tries to match the parameters you give to the parameters given T's methods. For each method, it writes down that the parameters in in the call are equal to their respective parameter in the method. If the equations are contradictory, the match fails and it tries another method. Otherwise, it succeeds and remembers the equations.

Here's an example. When we call Give "Mary" "Jill" "a red rose", the system tries to match the arguments to the first method. It gets:

- ?giver = "Mary"
- ?giver = "Jill"
- ?item = "a red rose"

The first two contradict one another, so we move on to the next method, which gives us:

- ?giver = "Mary"
- ?receiver = "Jill"
- ?item = "a red rose"

and that works. From that point on, the system remembers those equations.

Mary gave Jill a red rose.

But if we say:

Give “Mr. Boss” “Mr. Boss” “a nice raise”

We get:

Mr. Boss gave Mr. Boss a nice raise.

Which isn’t fluent. We can fix this, by having another method that specifically detects the case of someone giving something to themselves:

```
Give ?giver ?giver ?item: ?giver gave himself ?item
Give ?giver ?receiver ?item: ?giver gave ?receiver?item.
```

If you don’t tag a task with [randomly], then Step will always try its methods in the order they appear in the file. But because the first method lists ?giver for both its first and second parameters, it can only run when its first two parameters are the same. In the Mr. Boss example, they are, and so we get:

Mr. Boss gave himself a nice raise.

But in the Mary/Jill example, the parameters **can’t match** to that first method, and so that method **fails**. Step abandons that method and moves on to the next method, which works properly.

HANDLING PREFERRED PRONOUNS: SPECIALIZING METHODS TO SPECIFIC INPUTS

The example above brings up the issue that English, and many other human languages, “mark” certain words based on gender: most speakers say “himself” when the person referred to identifies as male, “herself” when the person identifies as female, “themselves” when the person/people referred to are unknown or a group of people, or otherwise take “them” as their preferred pronoun, and “itself” when the object in question either isn’t alive, or is a person who uses it as their preferred pronoun.

The code above uses the fixed text “himself” regardless of who the giver is. So if we’re generating text about Mary, then it’s going to refer to Mary with the masculine pronoun regardless of whether that’s their preferred pronoun or not.

We can fix this, by making the generation of the pronoun it’s own task. “Himself” is called a reflexive pronoun, so we’ll call the task Reflexive:

```
Give ?giver ?giver ?item: ?giver gave [Reflexive ?giver] ?item
Give ?giver ?receiver ?item: ?giver gave ?receiver ?item.
```

Now we can define Reflexive to generate the preferred pronoun for each person:

```
Reflexive “Mr. Boss”: himself
Reflexive “Mary”: herself
Reflexive ?: themselves
```

Notice that the first two of these methods specify **specific values** for their parameters. The first method will only match when the parameter is “Mr. Boss”. For any other value, it will fail to match and the system will move on to

the second method. But it will only match for the specific value, “Mary”. So we have Mary and Mr. Boss covered. The last method is a catch-all. If the parameter is anything else (? is a variable, remember), then the system will be conservative and print “themselves”.

Note that it’s important that the catch-all method come last, since the system tries methods in order (unless you told it to try them randomly, of course). It doesn’t matter what order the Mr. Boss and Mary methods are in, however, so long as they’re before the catch-all method.

A NICE SHORTHAND

You may have noticed by now that lines such as:

```
Give ?giver ?giver ?item: ?giver gave [Reflexive ?giver] ?item
```

can take some work to read. The formatting makes it somewhat easier, but the code is still messy and confusing. For a method like this, there isn’t a whole lot that can be done to improve it. But there is a useful shorthand that slightly improves this method and is quite useful for long passages of text with a few tasks embedded in them.

Here’s the shorthand: rather than saying “[Task ?variable]”, you can just say “?variable/Task”. So we can simplify the above method to:

```
Give ?giver ?giver ?item: ?giver gave ?giver/Reflexive ?item
```

Again, not a huge improvement. But it makes a difference when you’re writing big chunks of text.

MENTIONING

In fact, even just saying ?item on the right hand side, is actually shorthand for [Mention ?item]. So the method above is really shorthand for:

```
Give ?giver ?giver ?item:  
[Mention ?giver] gave [Reflexive ?giver] [Mention ?item]  
[end]
```

The only reason this matters is that it means you can control how things are printed by giving different methods for Mention. This is super useful, as we’ll talk about now.