AUD6204 - Programming Environments
Lesson 3.1

Basics 3

Contents

1 Data Storage **2**
- 1.1 Arrays . . . 2
- 1.2 Naming Arrays . . . 2
- 1.3 Arrays as GUI objects . . . 3

2 Send and Receive **6**
- 2.1 Send . . . 6
- 2.2 Receive . . . 6
- 2.3 Send/Receive Symbols . . . 6

3 Abstractions **7**

1 Data Storage

So far, we have (without explicitly stating it) been using basic objects to store data (for example, when we change a vslider/hslider it stores the new value inside it).

This is fine when we start, but soon becomes limiting (imagine trying to store a pattern 64 beats long - you would need 64 different objects!)

1.1 Arrays

In many programming environments, we use objects called arrays (see Table 1) to store lists of data. According to the PD help files:
"Arrays, in most programming environments, are considered to be "a sequence of objects all of which have the same variable type" wherein each object is called an element and the elements are numbered in a row: 0, 1, 2, 3, etc. These numbers are called indices. Each index holds a corresponding value."

Position/Index	0	1	2	3	475	6	7	8	9
Stored Value	0.1	0.567	0.134	1.1	6	0.234	0.01	3.14157	0

Table 1: An example array. Note how the first position has an index of 0! So if I asked the computer to tell me the value at index 2 of this array it would give me back a value of '0.134'

Think of it kind of like lockers at a train station. Each locker has a number. In order to store information we first need to choose a locker (i.e. an index in the array) and then put the information we want to store inside. If we want to retrieve that piece of information we just go back and look in the locker number (i.e. index) and look inside.

1.2 Naming Arrays

When we create an array we need to give it a name, this is because the computer needs to dedicated some section of its memory to store the array in and the name is our way of accessing that section of memory to write to/read from the array. Returning to our locker example from earlier, if you just know the locker number that's not enough - it could be in any train station in the city, so we first need to know which station it is in (the name of the array), and then which locker (the index).

1.3 Arrays as GUI objects

Not only do arrays allow is to store and retrieve large amounts of data, but they can also act as a user interface - allowing us to essentially create multiple sliders within one object. NOTE: Edit mode needs to be turned **off** to interact with the contents of the array.

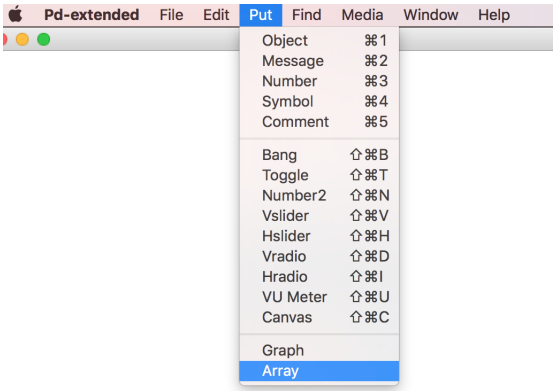

Figure 1: In PD, we **create** an array using the "Put" menu → **Put** >>> **Array**

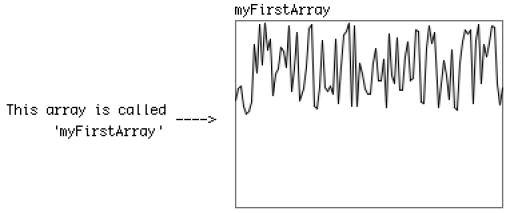

Figure 2: The array is displayed as a graph in our patch

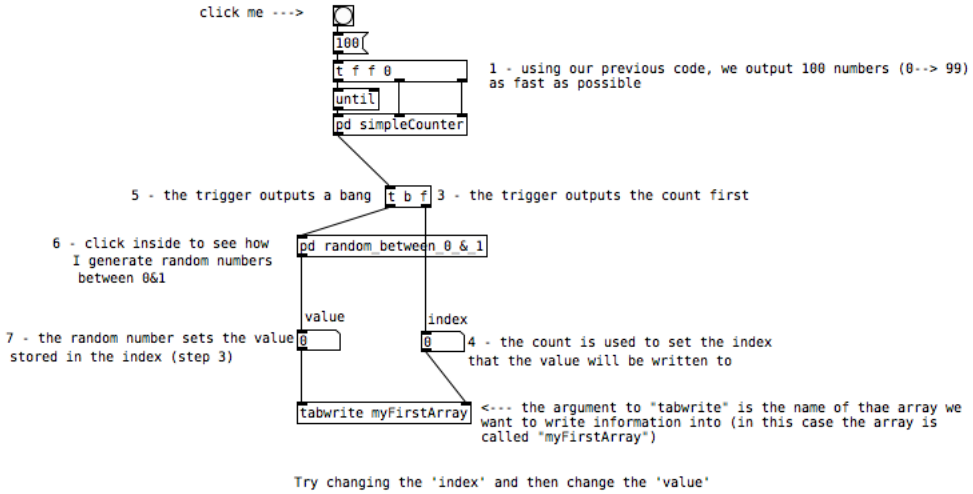

Figure 3: We **store** data in an array using **tabwrite**

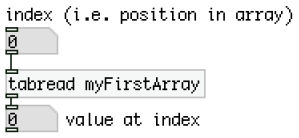

Figure 4: We **read** data from an array using **tabread**

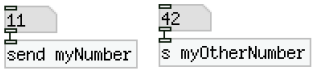

Figure 5: "**send** " objects in PD. Note that each object has a name, and 's' can be used as shorthand.

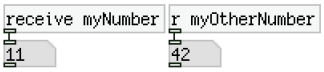

Figure 6: "**receive** " objects in PD. Note that each object has a name, and that 'r' can be used as shorthand

2 Send and Receive

Up until now we have been sending information from one object to another using patch cables, but this can get messy fast. Fortunately there are other ways to get data from one place in your patch to another.

The most common example is to use "**send**" and "**receive**" objects.

2.1 Send

Unsurprisingly, **send** objects (see Fig. 5) take the data they get in their inlets and send it to other places in your patch, or even in other patches/subpatches. (When creating a "**send**" object we must give it a name!!!).

2.2 Receive

"**receive**" objects, unsurprisingly, receive and output any information that is input into a send object with *the same name* (see Fig. 6).
Multiple send/receive objects may use the same name.

2.3 Send/Receive Symbols

In PD we can go one step further, and set a send or receive symbol for some interface object through their "Properties" menu (see Fig. 7).

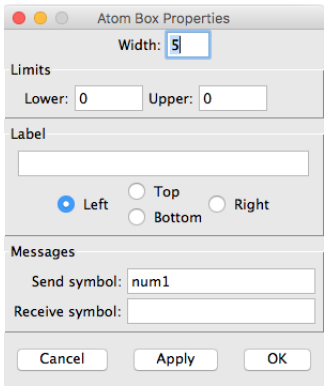

Figure 7: The "Properties" window of a number box. Note how the "Send symbol" has been changed to *num1*.

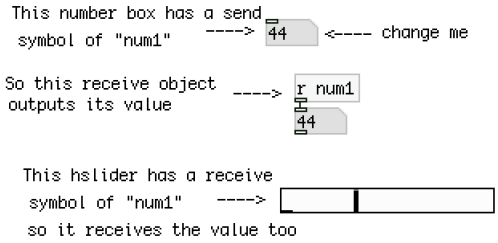

Figure 8: An example of sending and receiving values using send and receive symbols.

This means we can send data directly from and receive data directly into objects, with out needing any cables (see Fig. 9)

3 Abstractions

We have already looked at how creating subpatches can be helpful for keeping our code tidy, and allowing us to reuse ideas. We can go a step further and, when we have a piece of code that is particularly useful we can simply save it as a new patch and then load that code into another patch as if it is a normal object. We call these patches abstractions.

Abstractions act just like normal objects, in that you can give them arguments and make multiple copies of them in one patch etc. The only things we need to be aware of is that when trying to load an abstraction it needs to be in the same folder as the patch that we are trying to load it into, and anytime we change the code in our abstraction it changes

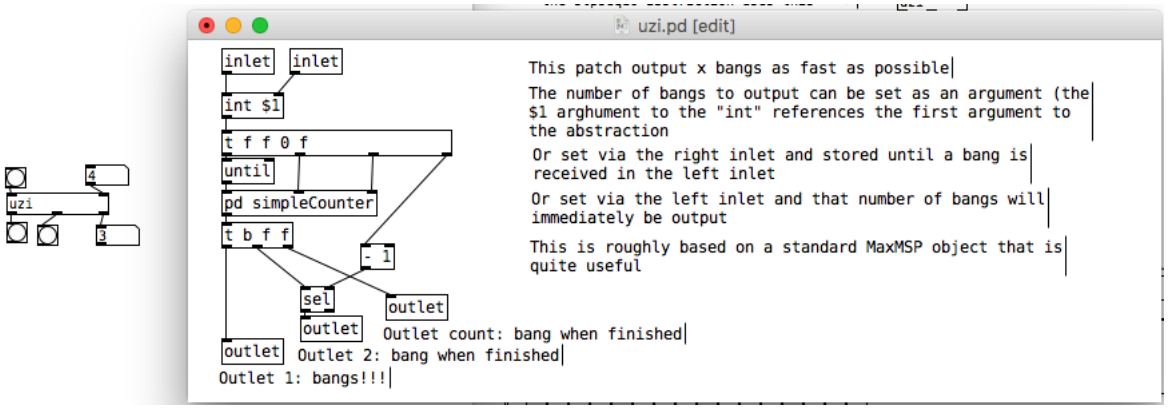

Figure 9: An example of a PD abstraction. You can see the patch in which the abstraction is loaded behind the "uzi" window.

everywhere that that abstraction has been loaded (the is both good, and a little dangerous as we can easily find code that has worked for ages suddenly stops because we have changed an abstraction.