The latest version of this doc is probably at
https://commonpike.github.io/nl.kw.processing.mods/dist/ProcessingMods.pdf
This is a work in progress. Interfaces may still change. If you have comments or suggestions,, let me know.

Processing Mods

https://github.com/commonpike/nl.kw.processing.mods

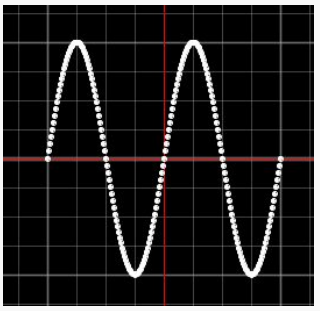
pike*20181121

What they do

Basics

A processing Mod is a java object with several ports. You can `get()` and `set()` values on ports, and the values you `get()` from a port may be updated every time you have `set()` a new value on another port. It is much like a regular function:

```
void draw() {
      Mod sin = new ModSin();
      for (int i=-100; i<= 100; i++) {
            sin.set("tick",i);
            point(i,sin.get("out"));
      }
      noLoop();
}
```

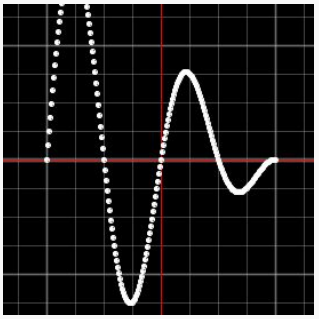


(NB: you can *easily* do the above without Mods in Processing)
(NB2: the graphics shown have a bit more code in them to get this nice preview [1])
(NB3: the above code is exactly the same as “Mod sin = new ModSin(); sin.plotter().plot(this)”)

The above code uses two ports of the ModSin mod, “**tick**” and “**out**”. ModSin also has ports “**amp**”, “**phase**”, “**speed**” and “**shift**”. Let’s see what happens if we play with amp, too:

```
void draw() {

 Mod sin = new ModSin();
  for (int i=-100; i<= 100; i++) {
    sin.set("tick",i).set("amp",100-i);
    point(i,sin.get("out"));
  }
  noLoop();
}
```



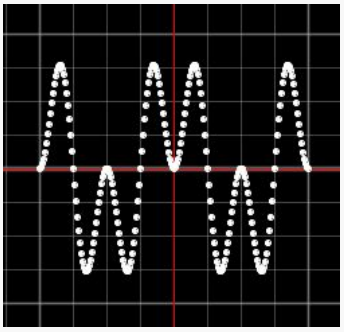
[1] try "`translate(width/2,height/2); scale(1,-1);`" to normalize your graphs like this..

Push and pull

The cool thing about Mods is that you can connect them - use the output of one mod as the input of another mod. Obviously, you can just do that in your own loop:

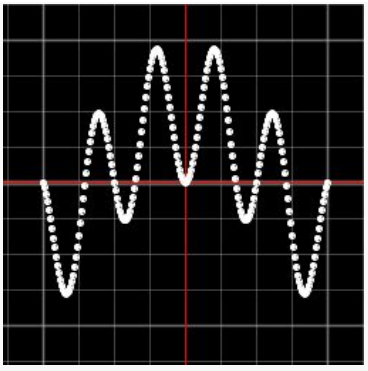
```
void draw() {

  Mod sin1 = new ModSin();
  Mod sin2 = new ModSin();
  for (int i=-100; i<= 100; i++) {
    sin1.set("tick",i*2);
    sin2.set("tick",i).set("amp",sin1.get("out"));
    point(i,sin2.get("out"));
  }
  noLoop();
}
```



But there are ways to connect mods together so that they run as one, by `pushing` or `pulling` ports of one mod to and from ports of the other mod:

```
void draw() {
   // connect the mods
   Mod sin = new ModSin();
   Mod subsin = new ModSin();
   sin.port("tick").push(subsin.port("tick"));
   sin.port("amp").pull(subsin.port("out"));
   subsin.set("speed",sqrt(20000));
   // now loop the main mod only
   for (int i=-100; i<= 100; i++) {
    sin.set("tick",i);
    point(i,sin.get("out"));
  }
  noLoop();
}
```

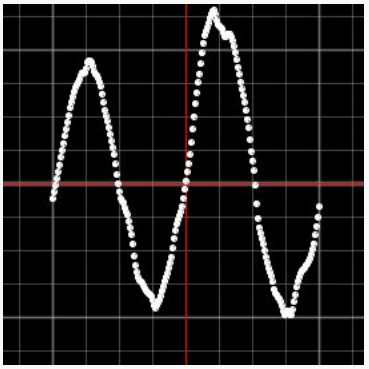


(NB: to get an updated value from a mod, you must either *get* or *pull* one of its ports; a *set* or *push* will not update its port values)

Chain and prechain

The particular type of setup where one port sends its source value to another mod and then retrieves a ‘modulated’ result value from that same mod is called a ‘chain’. You can do this using `push` and `pull` yourself, but also via the shorter `chain` method:

```
void draw() {
  Mod sin = new ModSin();
  Mod noise = new ModNoise(this);
  sin.port("out").chain(noise,"in","out");
  for (int i=-100; i<= 100; i++) {
    sin.set("tick",i);
    point(i,sin.get("out"));
  }
  noLoop();
}
```



(NB: ModNoise needs `this` as an argument to the constructor. If you omit this, it will complain, and then fail)

The line

```
sin.port("out").chain(noise,"in","out");
```

is exactly the same as

```
sin.port("out").push(noise,"in").pull(noise,"out");
```

But with `chain`, you can chain a second Mod on the same `port`, and it will be added on top of the first one - which would be harder to do using just the `push` and `pull` methods:

```
sin.port("out").chain(noise,"in","out").chain(median,"in","out")...;
```

`Prechain` does almost the same. The above line is effectively the same as

```
sin.port("out").chain(median,"in","out").prechain(noise,"in","out")...;
```

As you probably expected by now, you can turn such clusters of Mods into a new Mod. Which you can then connect to other Mods. Which you can turn into new Mods. Ad infinitum. For creating your own mods, see creating your own mods.

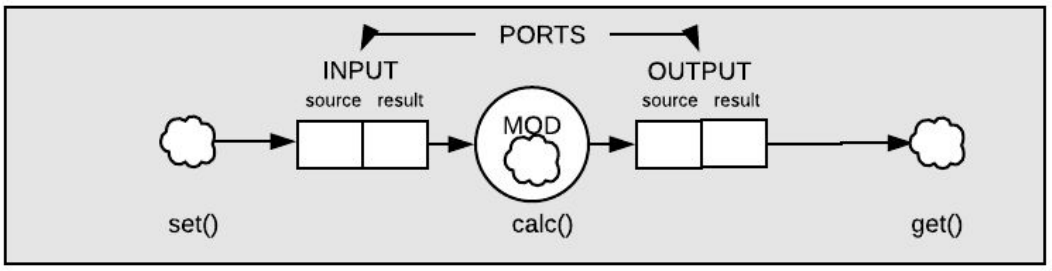
How they do it

A processing Mod is a java Object with several ModPorts.

- Any time you `set` (or `push`) a value on a port it is just stored there.
- Any time you `get` (or `pull`) a value from a port, the mod will examine all its ports and if necessary `calc()` new results before it returns its value.
- It is the `calc` method that defines the behaviour of the Mod.

NB: notice how it's necessary to `get` or `pull` a Mod to trigger it to recalculate its values.

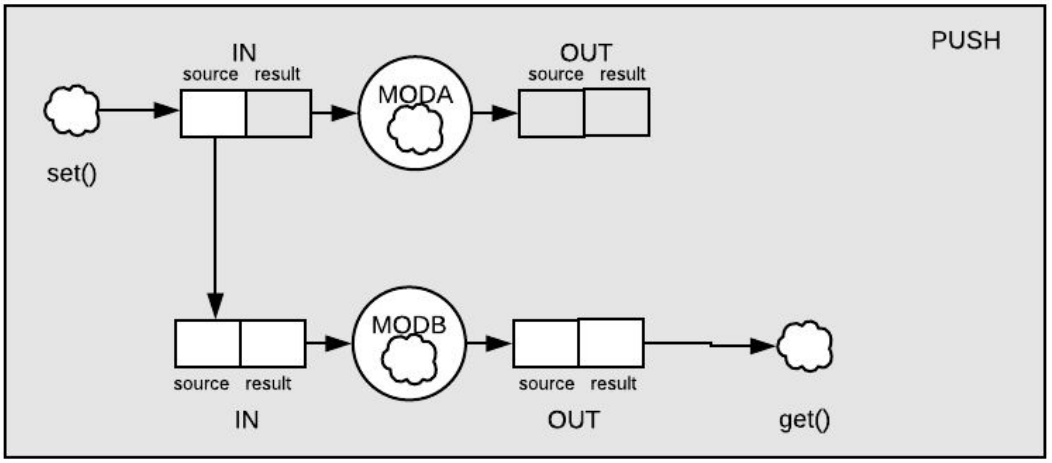
Below is a theoretical sketch of a Mod with two ports: input and output.



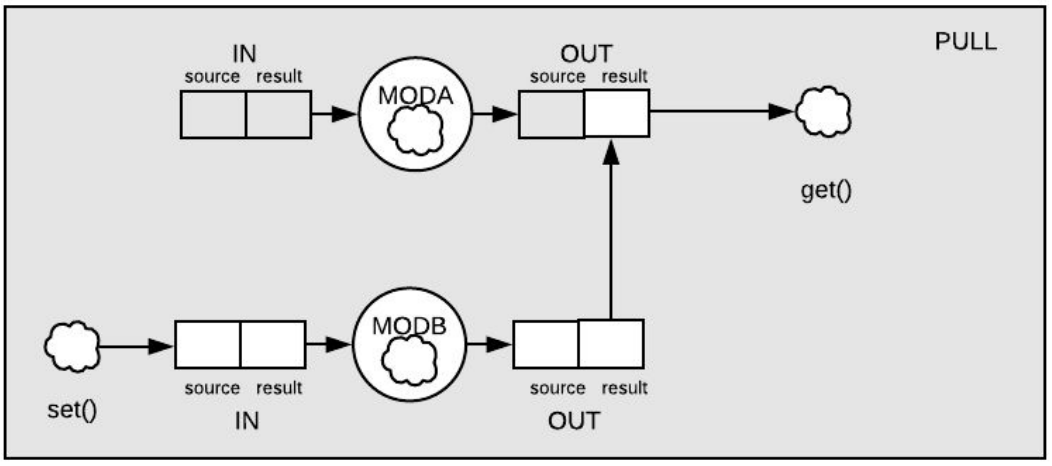
Each port has a `source` and `result` part. Whenever you `set()` a value, it is set on the source part. Whenever you `get` a value, the result part is returned. By default, when you try to `get` a value, the source part is simply copied to the result part and then returned.

Each port can have exactly one `push` port assigned, and exactly one `pull` port.

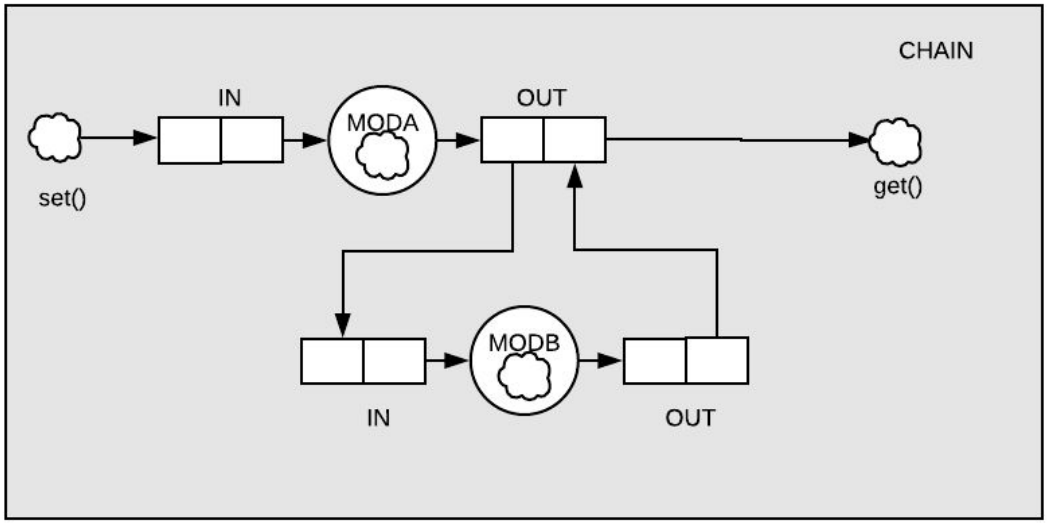
If the port has a `push` port assigned, anytime you `set` a value on that port, it will also copy that value to the source part of its push port:



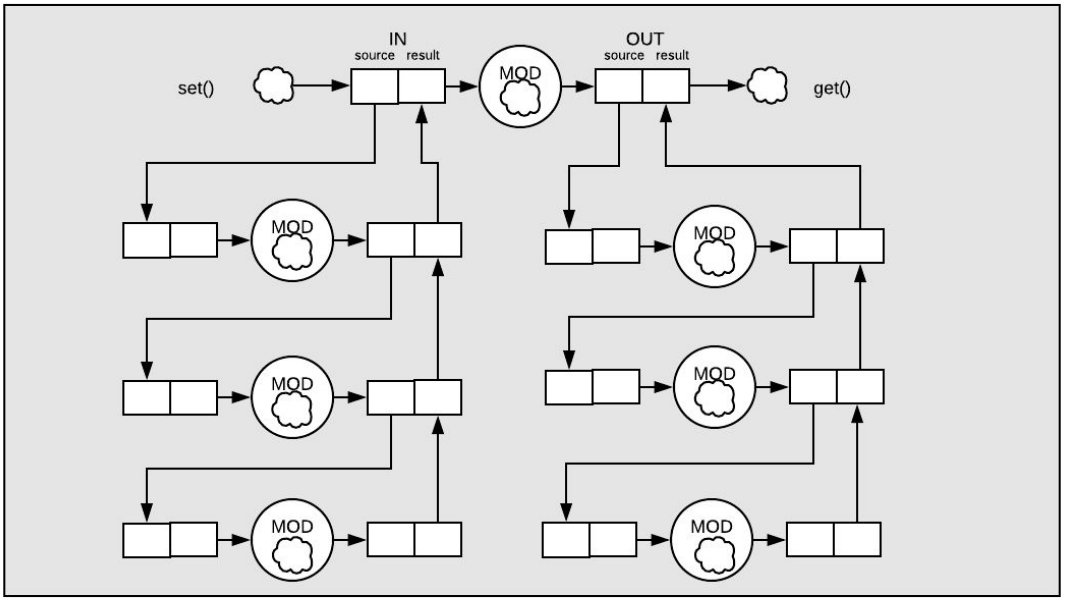
If a `pull` port is assigned to the port, whenever you `get` a value from that port, it ignores its own source part. Instead, it copies the result part of the `pull` port to its own result part, and then returns that:



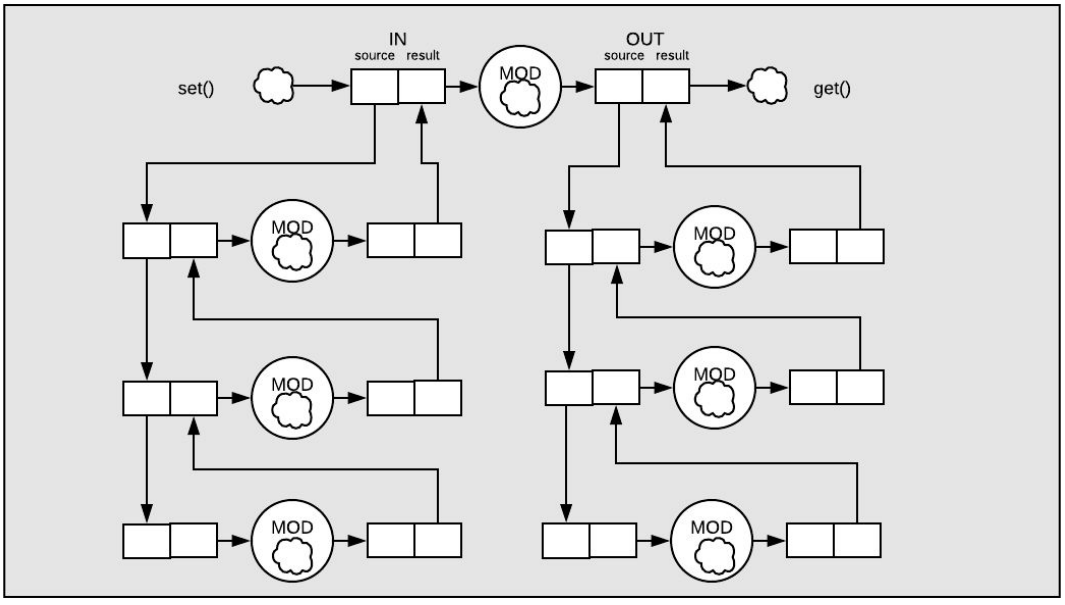
This becomes more interesting if you combine push and pull to 'hook' a port to another Mod, which is what 'chain' does:



Just for the heck of it, here are some chains stacked on top of eachother:



And `prechain` only works slightly different. Here are some prechains stacked in front of eachother - try to figure out the difference:

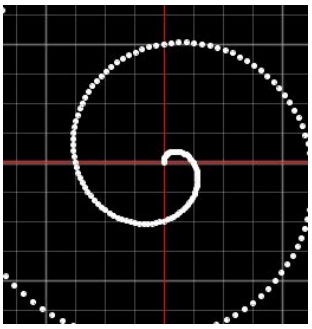


Obviously, you can combine `push`, `pull`, `chain` and `prechain` as needed, as long as they don't conflict with each other.

Conventions

Ranges and domains

In general: ports are doubles, ranging from -100 to 100 and defaulting at either 0 or 100.



To be able to connect ports from one mod to another, they should all more or less use the same output and input ranges. By default, ports return doubles between -100 and 100, like a signed percentage, and expect similar values in return. The default value of a port is usually either 0 or 100. For example, you may have noticed ModSin above, with a default speed of 100, returns a single sine wave with tick ranging between 0 and 100, while returning values between -100 and 100.
Not all Mods limit themselves to that range. ModSpiral for example spirals nicely between 0 and 100 in the first 100 ticks, but then it just spirals way out of bounds..

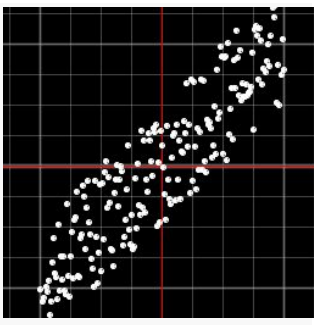
Input and output ports

Between mods, the port names vary. ModSin, which you've seen, has one output port (`**out**`) and arguably one major input port (`**tick**`). The other ports are `**amp**` (amplitude - vertical multiplication), `**speed**` (a horizontal multiplication), `**shift**` (a vertical translation) and `**phase**` (a horizontal translation). These are common port names.

ModSin does not really have an 'in' port, because it just depends on how you want to use it what port you would send your 'input' to.

Not all mods have a `tick` port. ModFuzz, for example, just adds 'amp' amount of randomness to `in` and returns that in `out`. There is really nothing to `tick` about that. It does however have a clear 'in' port:

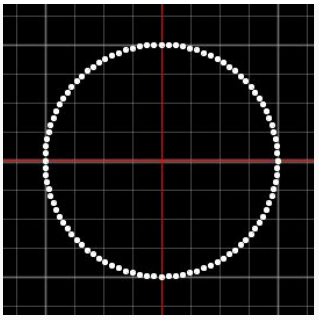
```
void draw() {
 Mod f = new ModFuzz();
  for (int i=-100; i<= 100; i++) {
    f.set("in",i);
    point(i,f.get("out"));
  }
  noLoop();
}
```



There may be more than one 'output' port. All ‘Mod2d*’ Mods have at least two output ports named `**outx**` and `**outy**`, for obvious reasons:

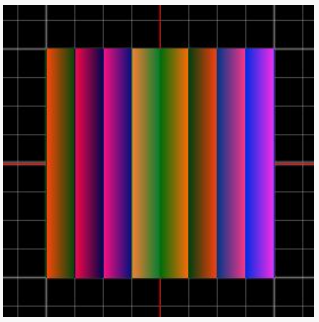
```
void draw() {

 Mod c = new Mod2dCirc();
  for (int i=-100; i<= 100; i++) {
    c.set("tick",i);
    point(c.get("outx"),c.get("outy"));
  }
  noLoop();
}
```



There is not even always a useful output port. When using ModColor, for example, you are mostly interested in its custom method named `color()`, which sets and gets a color value:

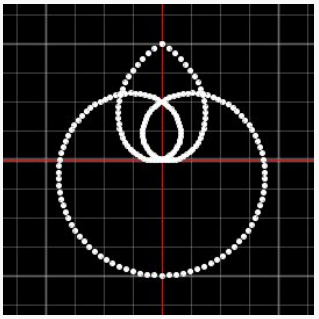
```
void draw() {
 ModColor c = new ModColor();
 for (int i=-100; i<= 100; i++) {
    if (i%25==0) {
      c.setColor(-random(255*255*255));//dontask
      c.set("green",c.get("green")/2);
    }
    c.set("red",int(i%25)*4);
    stroke(c.getColor());
    line(i,-100,i,100);
  }
  noLoop();
}
```



Subports

Mods that combine other mods can assign names to such 'submods' for your convenience. Mod2dSpiral, for example, uses ModLin for its growing speed, and has called it "grow". Below are two examples on how to set or get a value on a port on its 'grow' submod:

```
void draw() {
  Mod s = new Mod2dSpiral();
  s.mod("grow").set("shift",100);
  for (int i=-100; i<= 100; i++) {
    s.set("grow","speed",-2*i);
    s.set("tick",i);
    point(s.get("outx"),-s.get("outy"));
  }
  noLoop();
}
```



So exactly what ports a Mod has just depends on how it works and what it does. To find out more info about your mod, use mod.report():

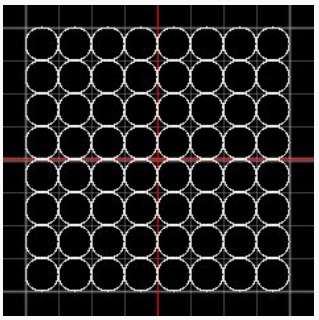
```
void draw() {
  Mod sin = new ModSin();
  sin.addMod("fuzz",new ModFuzz());
  sin.port("out").chain("fuzz","in","out");
  print(sin.report());
  noLoop();
  /*
   Mod ModSin
     Port ModSin.phase:      src 0.0  res 0.0
     Port ModSin.shift:      src 0.0  res 0.0
     Port ModSin.out:        src 0.0  res -5.496292  push: fuzz.in  pull: fuzz.out
     Port ModSin.amp:        src 100.0 res 100.0
     Port ModSin.tick:       src 0.0  res 0.0
     Port ModSin.speed:      src 100.0 res 100.0
     Mod fuzz
       Port fuzz.in:          src 0.0  res 0.0
       Port fuzz.out:        src -5.496292  res -5.496292
       Port fuzz.amp:        src 100.0  res 100.0

  */
 }
```

Plots, points and shapes

Instead of looping and plotting the output of a mod yourself, you can use a ModPlotter. It requires 3 ports: the `in` port, the `outx` port and the `outy` port. You can pass a range (to animate port in on) and a domain (to which outx and outy are scaled and translated):

```
void draw() {
  Mod c=new Mod2dCirc();
  ModPlotter p=c.plotter("tick","outx","outy");
  p.range(0,100,1);
  for (int x=-4; x=4; x++) {
    for (int y=-4; y<=4; y++) {
      p.domain(x*25,y*25,(x+1)*25,(y+1)*25);
      p.plot(this);
    }
  }
  noLoop();
}
```

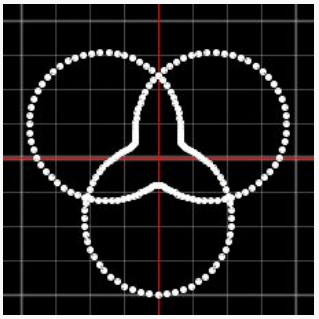


Some mods provide nice defaults for the Mod, so you don't have to specify ports. Range defaults to -100, 100, as do both domains; the output ranges (that is, the expected source values of the used output ports, not shown here) can also be specified, and also default to -100,100.

The command `p.plot(this);` tells the plotter exactly on which output to plot - this one.

The ModPlotter can also return all the coordinates as an array-of-arrays of doubles:

```
void draw() {
  Mod c=new Mod2dCirc();
  Mod s = new ModSin();
  s.set("speed",150).set("shift",60).set("amp",40);
  c.port("tick").push(s,"tick");
  c.port("radius").pull(s,"out");
  double[][] points = p=c.plotter().points();
  for (int pc=0; pc<points.length;pc++) {
    point(points[pc][0],points[pc][1]);
  }
  noLoop();
}
```

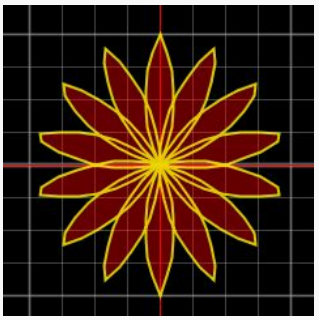


And finally, a ModPlotter can create a PShape for you which you can reuse in your own code. By default it is closed, so you can use a fill color, too :

```
void draw() {
  Mod c = new Mod2dCirc();
  Mod tri = new ModTri();
  tri.set("speed",350);
  c.port("tick").push(tri,"tick");
  c.port("radius").pull(tri,"out");

  fill(0x66ff0000);
  stroke(0xccffff00);
  strokeWeight(2);
  shape(c.plotter().shape(this));

  noLoop();
}
```



Again `p.shape(this);` tells the plotter exactly which output to create shapes for.

Creating your own Mods

Creating your own Mod is easy: just extend an existing Mod (or mod itself), optionally add some ports, and optionally implement the calc() method:

```
class ModNop extends Mod {
   ModNop() {
     super();
     addPort("in").def(0);
     addPort("out").def(0);
   }
   protected void calc() {
     set("out",get("in"));
   }
}
```

If you extend a mod, maybe you don't even have to implement calc() yourself:

```
class Mod2dSpiral extends Mod2dCirc {
  Mod2dSpiral() {
    super();
    addMod("grow",new ModLin());
    port("tick").push("grow","tick");
    port("radius").pull("grow","out");
  }
}
```

This says: a spiral is a circle, but it has a lineair 'grow'. The grows ticks along with the spiral, and the output of the grow goes to the radius. The `calc()` stuff happens in Mod2dCirc.

If you want to implement the default input and output ports for the plotter, implement the `plotter()` method:

```
class ModFuzz extends Mod {

   ModFuzz() {
     super();
     addPort("in").def(0);
     addPort("amp").def(100);
     addPort("out");
   }

   protected void calc() {
      double in = get("in");
      double amp = get("amp");
      double out = in+random(amp)-amp/2;
      set("out",out);
   }

   public ModPlotter plotter() {
    return plotter("in","in","out");
   }
}
```

Basic Methods

Below are most basic methods. These are often shorthand notation to deeper methods, and there will be variations on the same methods using objects instead of strings, etcetera. Most of the setters return the objects on which values were set, so they can be chained (like *mod1.addMod("mod2").set("x",y).get("q")*).
For the exact and full API, read the JavaDoc. Otherwise, play around !

```
Mod foo = new ModFoo();

// for creating new mods
foo.addPort(String portname); // add a port
foo.addMod(modname); // register a submod for easy access

// submods and ports
foo.mod(modname); // retrieve a submod
foo.port(portname); // retrieve a port
foo.port(modname,portname); // retrieve a port from a submod

// getters and setters
foo.get(portname); // get value from a port
foo.get(modname,portname); // get value from a port of a submod
foo.set(portname,portvalue); // set value on a port
foo.set(modname,portname,value); // set value on a port of a submod

Mod bar = new ModBar();

// port send and receive
foo.port(portname).push(bar,pushname);
foo.port(portname).pull(bar,pullname);

// port chain and prechain
foo.port(portname).chain(bar,sendname,receivename);
foo.port(portname).prechain(bar,sendname,receivename);

// plotter and shapes
foo.plotter(in,outx,outy);
foo.plotter().plot();
foo.plotter().shape();
```

The latest version of this doc is probably at
https://commonpike.github.io/nl.kw.processing.mods/dist/ProcessingMods.pdf
This is a work in progress. Interfaces may still change. If you have comments or suggestions, let me know.