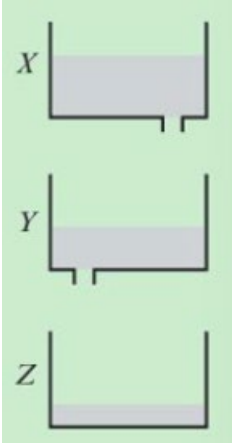
Draining Tanks

A common problem in differential equations involves water tanks which drain into one another. The sketch in this project was inspired by one such problem, the details of which are given below:

Three tanks, holding 300, 200 and 100 gallons of water respectively, drain into one another. The first tank drains into the second at a rate of r gallons per hour, where r is the amount of water in the tank at that moment. Find expressions in terms of t for the amount of water in each of the three tanks, and hence find how long it will take for the first two tanks to hold the same amount.
Later, a second tap is attached to the first tank, doubling the rate at which it drains. If the first tank initially contains 600 gallons, and the other two are empty, when is the amount in the second tank at maximum? When will the last two tanks hold the same amount?



The maths

These problems can be tackled directly, by forming and solving differential equations:

$$\frac{dx}{dt} = -x \qquad \frac{dy}{dt} = x - y \qquad \frac{dz}{dt} = y$$

But we could also simulate the situation by reproducing in code the physical scenario. In just the same way that we simulate the complex effects of gravity by simply adding an acceleration to our velocity, then the velocity to our position, we can keep track of small changes in x, y or z for a suitably small change in t, and *analyse the situation numerically* rather than analytically. Incidentally, this is the essence of what computers generally do when asked to 'solve' a differential equation.

What we want

Just like when tackling a maths problem, it helps to think about what we're after before diving in head-first to our imaginary water-tanks. For starters, I'd like a quick way to visualise a tank with different amounts of water in it.

Visually it should include:

- A boundary (sides and a base) at a given position with given dimensions.
- Water (eg blue rectangle within, at a height which varies according to the volume).

Behind the scenes it will need to store information about:

- Volume (how much is currently in it).
- Capacity (maximum amount it can hold).

It should also incorporate, for ease of use, the following functionality:

- A way to add (or remove) a given amount of water.
- A way to set the volume to a given value.

The tank class

```
class Tank{
  constructor(x, y, w, h, capacity){
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.capacity = capacity;
    this.volume = 0;
  }

  add(amount){
    this.volume += amount;
    this.volume = min(this.volume, this.capacity);
    this.volume = max(0, this.volume);
  }

  set(amount){
    this.add(amount - this.volume);
  }

  display(){
    this.drawContents();
    this.drawContainer();
    this.drawLabel();
  }
```

The constructor takes care of the position (I'm treating *x, y, w* and *h* in the same way that *rect* does, with *x* and *y* defining the top left corner, and *w* and *h* the width and height of the container respectively.

The capacity is set at the start, but to keep things simple I'm making the current volume zero. This can be set directly using the *set* method, so it doesn't have to be done immediately upon initialization.

The *add* method simply adds the given amount to the current volume (note that *amount* could be negative, so this works for both increasing and decreasing the amount). I'm using *min* and *max* to force the volume to stay between 0 and the given capacity. Notice that, by using *add* in my *set* method, I avoid having to duplicate this.

Finally, I've included a *display* method, but I've left the details till later. I'll want the contents drawn in first, so that the container isn't covered by the contents, then the container itself, and finally some kind of label (it could be helpful to see the actual volume at any given moment).

The draw methods

The *drawContainer* method will be fairly straightforward: it doesn't depend on a changing volume, for a start. We can use the *beginShape()* and *endShape()* built-in functions to construct it as a series of connected vertices:

```
drawContainer(){
  stroke(0);
  strokeWeight(2);
  noFill();
  beginShape();
  vertex(this.x         , this.y         );
  vertex(this.x         , this.y + this.h);
  vertex(this.x + this.w, this.y + this.h);
  vertex(this.x + this.w, this.y         );
  endShape();
}
```

Remember not to make any assumptions about the current *stroke* and *fill* settings, resetting them whenever you make a shape.

My four vertices are the top left, bottom left, bottom right and top right, in that order. By not using the keyword *CLOSE* or including the first vertex again at the end, I'm constructing an open-topped tank.

The *drawContents* method takes a little more thought, since we have to consider the appropriate depth to draw the rectangle. However, given our pre-set capacity, this is a simple matter of using *map*:

```
drawContents(){
  noStroke();
  fill("lightblue");
  let depth = map(this.volume, 0, this.capacity, 0, this.h);
  rect(this.x, this.y + this.h - depth, this.w, depth);
}
```

This is the water, so we'll make it blue. Note that the temporary variable *depth* makes the final line more readable, and hopefully easier to error-check and verify.

The *drawLabel* method is simpler still. Just below the tank, in the middle, we display the volume:

```
drawLabel(){
  noStroke();
  fill(0);
  text(round(this.volume, 0), this.x + this.w / 2, this.y + this.h + 10);
}
```

I'm using *round* instead of *int* so that values like 499.95 are displayed as 500, not 499.

Making a tester tank

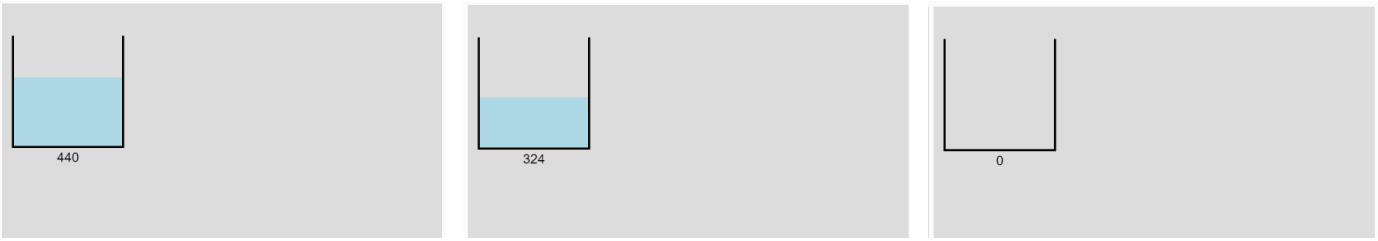
Now we've put together what it means to be a tank, it's time we tested it out by making one:

```
let X;

function setup() {
  createCanvas(400, 400);
  textAlign(CENTER, CENTER);
  X = new Tank(10, 30, 100, 100, 700);
  X.set(500);
}

function draw() {
  background(220);
  X.display();
  X.add(-1);
}
```

In order to access and modify this tank it needs to be declared outside the *setup* and *draw* functions, and then we define its parameters in *setup* (including setting the initial volume to 500).

Note that I've also included the *textAlign* statement to ensure our label is centrally placed below the tank.

In *draw* we simply display the tank, and also, every frame, remove 1 unit of volume from it. We'll improve on this rough and ready frame-by-frame method shortly.



The tank should 'drain' pretty fast (usually p5 sketches run at around 60 fps, so it'll take under 10 seconds).

Making one tank drain into another

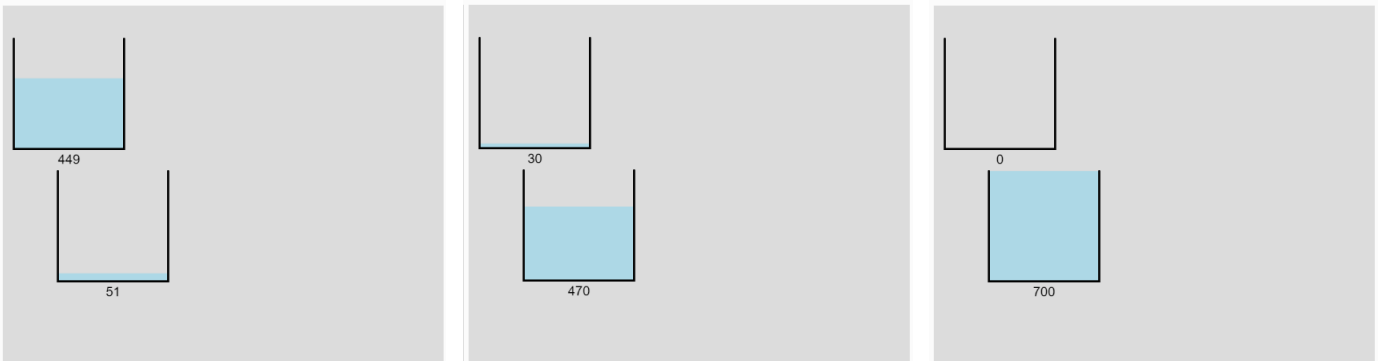
The task of making a tank is done: making a second is much easier. And we can see about animating drainage from one into the other, as well as making the rate of drainage more precise.

```
let X, Y;

function setup() {
  createCanvas(400, 400);
  textAlign(CENTER, CENTER);
  X = new Tank(10, 30, 100, 100, 700);
  Y = new Tank(50, 150, 100, 100, 700);
  X.set(500);
}

function draw() {
  background(220);
  let dx = -1;
  let dy = 1;
  X.display();
  Y.display();
  X.add(dx);
  Y.add(dy);
}
```

My second tank, which I've called Y, will sit below, and slightly to one side of, the first tank. If I were making loads of these, I might put them in an array, but we'll only have two or three, so I'm content to have a few duplicate lines of code.

Notice that, since I didn't use *set* with tank Y, it will have the default initial volume of 0.

In *draw* I've gone one step closer to a more formal and rigorous way to define drainage, by defining variables *dx* and *dy* which determine the rate at which the two tanks change their volume. Currently tank X loses 1 gallon per frame, and tank Y gains 1 gallon. But there's something of a flaw in this system, which you can see if you let it run to the end:



Forget conservation of energy – we don't even have conservation of water! While the first tank is draining, the second appears to collect whatever the first loses, but once the first is empty, the second carries on filling regardless! This is a flaw which will be readily corrected when we take into account the true driver of the rate of flow: the actual amount in the first tank.

Writing in the differential equations

In the original question that our animation is based on, each tank drains at a rate proportional to its own volume, and with a proportionality constant of 1 as well, so that $\frac{dx}{dt} = -x$. In other words, when tank X holds 300 gallons, at that instant it will be draining at a rate of 300 gallons an hour. We need a *dt* term…

```
let X, Y;
let dt = 0.01;

function setup() {...}

function draw() {
  background(220);
  let dx = -X.volume * dt;
  let dy = X.volume * dt;
  X.display();
  Y.display();
  X.add(dx);
  Y.add(dy);
}
```

Nothing changes in *setup* (I've minimised it for now), but I've added a global variable *dt* which represents the time-step we increment by each frame.

Notice that the equivalent of $\frac{dx}{dt} = -x$ is $dx = -x\,dt$, where, since *dx* and *dt* are no longer infinitesimals, but just very small values, I can compute *dx* directly and use it to update tanks X and Y.

You should find now that the top tank drains quickly at first, but ever more slowly as it empties into the bottom tank (which only ever gains water if the other tank loses it).

Visualising the rate of flow

Drawing a tap seems like too much work, but we can easily make a blue line showing the path of water from one tank to the other. We could just draw a line, and set the *strokeWeight* proportional to the flow, but to avoid weird rounded ends I'm going to use a very thin rectangle:

```
function draw() {
  background(220);
  let dx = -X.volume * dt;
  let dy = X.volume * dt;
  X.display();
  Y.display();
  X.add(dx);
  Y.add(dy);
  let flow = 0.02 * dy/dt;
  noStroke();
  fill("lightblue");
  rect(X.x + X.w - flow - 5, X.y + X.h - 1, flow, 120);
}
```

With a bit of trial and error, I've settled on scaling down the flow rate (which, note, is $\frac{dy}{dt}$) by a factor of 0.02 so that the width of the resulting line is in proportion with the rate of flow, but not too ludicrously wide.

This could have been built into the tank class, but depending on the setup of tanks its parameters may change.

One more tank

We're now in a position to add in one final tank, and see the full simulation in all its leaky glory:

```
let X, Y, Z;
```

In the global scope, declare Z.

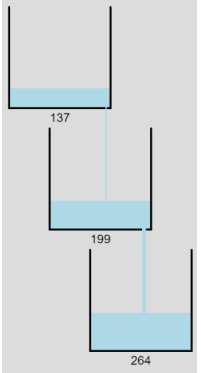
```
X = new Tank(10, 30, 100, 100, 700);
Y = new Tank(50, 150, 100, 100, 700);
Z = new Tank(90, 270, 100, 100, 700);
X.set(300);
Y.set(200);
Z.set(100);
```

In *setup*, add the third tank.

And set the initial values of all three.

```
function draw() {
  background(220);
  let dx = -X.volume * dt;
  let dy = (X.volume - Y.volume) * dt;
  let dz = Y.volume * dt;
  X.display();
  Y.display();
  Z.display();
  X.add(dx);
  Y.add(dy);
  Z.add(dz);
  let flow1 = 0.02 * dy/dt;
  let flow2 = 0.02 * dz/dt;
  noStroke();
  fill("lightblue");
  rect(X.x + X.w - flow1 - 5, X.y + X.h - 1, flow1, 120);
  rect(Y.x + Y.w - flow2 - 5, Y.y + Y.h - 1, flow2, 120);
}
```

And finally, in *draw*, update the differential equations that govern the behaviour, and add a second 'flow' indicator between the tanks.



Bonus: Keep track of the time so you can see when things happen. Use a slider for *dt* to change the speed.