Clock

The *hour*, *minute* and *second* functions built into p5 are asking to be made into a clock.

We're going to start with a simple digital clock, then look at how to construct an analogue clock face making use of angles and a little trigonometry.

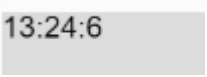
The basic version

The first thing to do is see how the built-in functions behave. This is probably the simplest way of creating a digital clock on screen:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  text(hour() + ":" + minute() + ":" + second(), 0, 10);
}
```

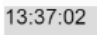
A perfectly serviceable digital clock in the corner of the screen with only one line of code.
In addition to the cosmetic modifications we may want to make, there's another slight issue:



Ideally we need each of these numbers to be 'padded' with a leading zero. Eg 13:24:06. If we're going to do this for all three numbers it's probably worth making a quick function:

```
function pad(num){
  if (num < 10){
    return "0" + num;
  }
  return num;
}
function draw() {
  background(220);
  text(pad(hour()) + ":" +
       pad(minute()) + ":" +
       pad(second()), 0, 10);
}
```

This basic padding is limited to 2 digit integers, but that's fine for our purposes. Note that I don't need to use *else*, because the only way the program will evaluate the *return num* instruction is if the condition above is not met.

This small addition fixes the problem: 

Notice that, for improved readability, it is common practice to press Enter and wrap long lines onto multiple lines. Since JavaScript is looking for the semi-colon to signal end-of-line, the computer won't get confused.

The OOP way

If you want your digital clock code to be readily portable to different sketches, it makes sense to embed it in its own object. When designing the class, consider how you might want to use the clock so you know what values to pass to the constructor function. Depending on the context in which you'll use it, you may well want to set the colour, for instance (foreground and background), and presumably size and location.

```
class DigitalClock{
  constructor(x, y, w, h, fg="black", bg="none"){
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.fg = fg;
    this.bg = bg;
  }
  pad(num){
    if (num < 10){
      return "0" + num;
    }
    return num;
  }
```

I'll cross the aspect ratio bridge when I come to it. For now, I think it would be helpful to define a rectangle within which the clock must sit, using *x, y, w, h* just like the *rect* drawing object. I've set default values for the foreground and background colours (although *"none"* isn't a valid colour, I intend to use this to see if a background is required at all).

Notice that I've also brought the *pad* function inside the class to make it a method bound to *DigitalClock*.

The *display* method

```
display(){
  let s = this.pad(second());
  let m = this.pad(minute());
  let h = this.pad(hour());
  let txt = h + ":" + m + ":" + s
  noStroke();
  if (this.bg != "none"){
    fill(this.bg);
    rect(this.x, this.y, this.w, this.h);
  }
  fill(this.fg);
  text(txt, this.x, this.y);
}
```

I'm generating the required text, making use of my *pad* method, and then if a background colour has been specified, I'm drawing a rectangle of that colour.
Remember to specify *noStroke()* and to set a *fill* value before rendering text.
Can you work out what's wrong with this naïve approach?

Working with text

There are a few really handy built-in functions to make our life easier here. Let's take them one at a time:

```
textAlign(CENTER, CENTER);
```

The first argument gives horizontal alignment, and the second gives vertical alignment. I'm adding this line just before rendering the text (so that my class code is portable to other sketches without requiring other modifications – I usually call this function just once in *setup* otherwise).

```
textStyle(BOLD);
```

You can set BOLD, or ITALIC, or BOLDITALIC.

```
textFont("Courier New");
```

And choose from the common fonts you might use.

```
let sf = this.w / textWidth(txt);
```

This handy function returns the width of a given text string, so we can readily compare with the available space, and scale accordingly.

```
textSize(textSize()*sf*0.9);
```

I don't even need to know precisely how font size relates to pixels on the screen (it's different for different characters anyway) – I can return the current size using the *textSize()* function with no arguments, and then pass the same function a value to scale it up so that it (almost) fills the available space.

```
text(txt, this.x + this.w/2, this.y + this.h/2);
```

14:17:08

With a slight modification, our numbers should now appear nicely spaced filling the available rectangle. Note that, since I'm aligning numbers centrally, I need to pass the *text* function the coordinates of the centre of my rectangle.

```
this.h = max(w/3, h);
```

With a bit of trial and error I've decided that the height of my clock needs to be at least $\frac{1}{3}$ the width, so I'll force that as a minimum at the start.

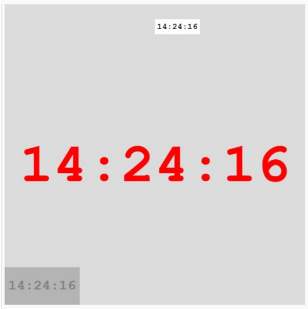
Testing our code

If you haven't already, make a clock or two by declaring some variables (or even an array), and making a few different ones to check that our code is robust enough to deal with a range of use cases:

```
let clocks = [];

function setup() {
  createCanvas(400, 400);
  clocks.push(new DigitalClock(0, height-50, 100, 50, "grey", 180))
  clocks.push(new DigitalClock(width - 200, 20, 60, 20, "black", "white"))
  clocks.push(new DigitalClock(width/2 - 200, height/2-50, 400, 20, "red", "none"));
}

function draw() {
  background(220);
  for (let clock of clocks){
    clock.display();
  }
}
```



Chasing the bugs

You might notice a bit of a judder with the clock, using the code described above. This is because different characters have slightly different widths, and because we were recalculating text size every time *display* was called. As long as the text is the right size for one time, it won't change significantly for others, so see if you can refactor your code a bit so it calculates the best font size once, and stores it as an attribute.

```
this.bg = bg;
this.formatText();
this.size = this.setFontSize("00:00:00");
```

An analogue clock

Now for the proper clock. As before, we'll make a class, but for simplicity I'll code in colours and font, leaving just the size and position for the user to define.

```
class AnalogClock{
  constructor(x, y, r){
    this.x = x;
    this.y = y;
    this.r = r;
    this.formatText();
  }
  formatText(){
    textAlign(CENTER, CENTER);
    textStyle(BOLDITALIC);
    textFont("Georgia");
    textSize(this.r/5);
  }
  display(){
    noStroke();
    fill("white");
    this.formatText();
    let s = second();
    let m = minute();
    let h = hour();

    for (let i=0; i<12; i++){
      let theta = 2*PI/12 * i;
      let x = this.r * cos(theta);
      let y = this.r * sin(theta)
      circle(this.x + x, this.y + y, this.r/40);
    }
```

Initially, we just need the coordinates of the centre and a radius for our clock face.

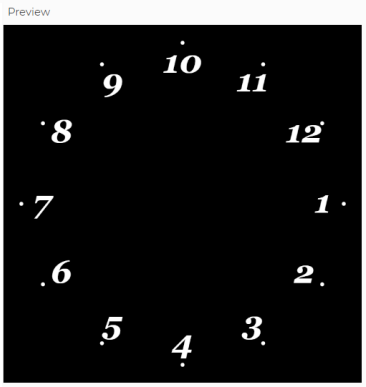
As hinted at in the previous section, we can separate out the text formatting, so that setting the size only happens once. Although in this case I'm trying to ensure sizes are all relative to the radius. With a bit of trial and error I'm setting *textSize* to a fifth of the radius.

In *display* we won't need to worry about padding: the seconds, minutes and hours will be used to determine the positions of the clock hands in a moment, but for now we can leave them.

We'll start by using trig to put twelve dots equally spaced around the circle. This is just $(r\cos\theta\,, r\sin\theta)$, translated to the centre of the clockface.

Now for the numbers:

```
for (let i=0; i<12; i++){
  let theta = 2*PI/12 * i;
  let x = this.r * cos(theta);
  let y = this.r * sin(theta)
  circle(this.x + x, this.y + y, this.r/40);
  text(i+1, this.x + 0.87*x, this.y + 0.87*y);
}
```



Something doesn't look quite right there…

Notice that the direction is correct – the inversion of the y-coordinate for on-screen graphics has taken care of that for us (the mathematical convention being to take anti-clockwise as positive), but we're starting in the wrong place. However, as long as we offset *theta* by an appropriate amount, we can correct this:

```
for (let i=0; i<12; i++){
  let theta = 2*PI/12 * (i - 2);
  let x = this.r * cos(theta);
  let y = this.r * sin(theta)
  circle(this.x + x, this.y + y, this.r/40);
  text(i+1, this.x + 0.87*x, this.y + 0.87*y);
}
```

Note also that I've tweaked the effective radius so that the numbers lie inside the clock face.

Now your clock should look more normal, it's time to work on the hands. Since these use very similar code, just with a different fraction, we'll try to do it with a single class method:

```
  this.displayHand(s, 1/60, this.r/100, this.r*0.8);
  this.displayHand(m, 1/60, this.r/50, this.r*0.7);
  this.displayHand(h, 1/12, this.r/25, this.r*0.5);

}
displayHand(value, fraction, thickness, length){
```

The three lines at the top here are called within the *display* method. I want to use these to set the parameters that will vary between the hands: the value the hand displays, but also the fraction of a full turn that each unit represents.
The other two parameters will represent the thickness of the hand and its length respectively.

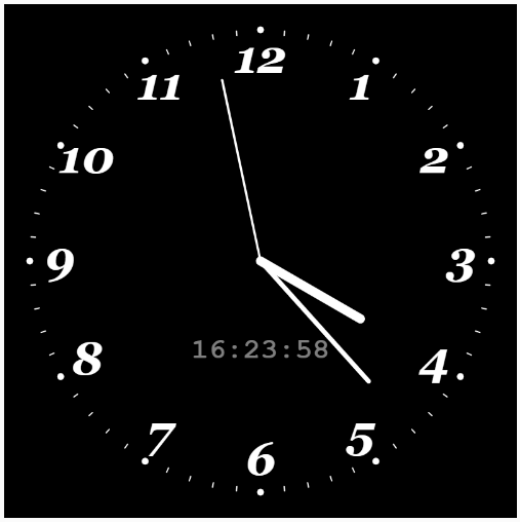
The *displayHand* method itself is quite short, but takes some careful thought to get right. You'll find it helpful to look back at how we displayed the twelve numerals around the clock face.

```
displayHand(value, fraction, thickness, length){
  let theta = (value * fraction - 3/12) * 2*PI;
  strokeWeight(thickness);
  stroke("white");
  line(this.x, this.y, this.x + length * cos(theta),
       this.y + length * sin(theta));
}
```

The clever bit here is to set the right value of *theta*. It will be determined by taking into account the fraction that each unit of value represents (eg at either 3 seconds or 3 minutes it'll be passed *3, 1/60*, so both will yield 0.05, whereas 3 hours will be passed in as *3, 1/12*, producing 0.25, or a quarter of a full turn.

Subtracting $\frac{3}{12}$ allows me to rotate my clock back to the default position like before, and then I multiply the fraction by a full turn in radians (2π). From there, it's just a case of drawing a line from the centre to the point $(l\cos\theta, l\sin\theta)$ where l is the length required.

And just like that, we have our clock! Combine it with one of your digital clocks to compare and make sure your hands aren't off:



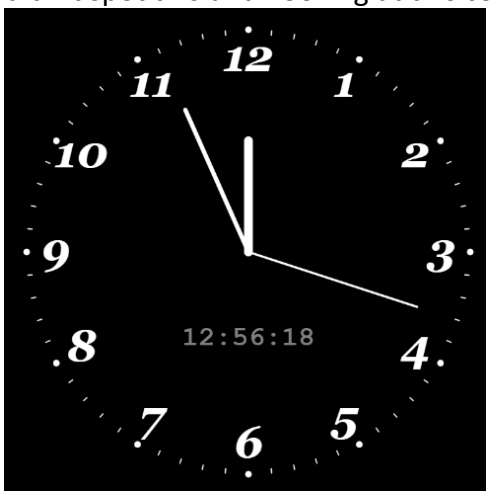
You'll notice that my clock incorporates lines at the one-minute marks as well as at 5 minutes. This is accomplished by making a separate loop for markers than we used for numbers:

```
stroke("white");
strokeWeight(1);
for (let i=0; i<60; i++){
  let theta = 2*PI/60 * i;
  let x = this.r * cos(theta);
  let y = this.r * sin(theta);
  if (i % 5 == 0){
    circle(this.x + x, this.y + y, this.r/40);
  }
  else{
    line(this.x + x, this.y + y, this.x + x*0.98, this.y + y*0.98);
  }
}
```

This code draws a circle at intervals of 5 minutes, but a line the rest of the time. The lines are also made so that they start at the edge of the defined circle but point towards the centre (using a scale factor to reduce their distance from the centre by 2%). I encourage you to experiment with this design and see if you can incorporate anything to make it a bit unique.

A curious quirk

I didn't spot this until looking at this code again just before $1pm$ – can you see what's wrong?



The second hand is ticking, which is fine for seconds, but generally we like the minute hand and, most importantly, the hour hand, to move smoothly from one number to the next. We can achieve this by passing the *displayHand* method a slightly different value for hours and minutes:

```
this.displayHand(s, 1/60, this.r/100, this.r*0.8);
this.displayHand(m+s/60, 1/60, this.r/50, this.r*0.7);
this.displayHand(h+m/60+s/3600, 1/12, this.r/25, this.r*0.5);
```

The value used to display the hour hand at 12:56:18 will no longer be 12, but rather $12 + \frac{56}{60} + \frac{18}{3600} \approx 12.94$, much closer to the 1 o' clock position.

Bonus: the function *millis()* is another in-built function like *second()* and *minute()*. With a couple of quick lines of code we can make the second hand move smoothly as well as the minute and hour hands.

In the clock's *constructor* method:

```
this.startSeconds = second();
```

Instead of using *second()* in the *display* method:

```
let s = millis()/1000 + this.startSeconds;
```