Functions and Recursion

DECO1012

Functions

```
int y = 0;

void setup() {
  size(300, 300);
}

void draw() {
  line(0, y, 300, y);
  y = y + 4;
}
```

Data type

Variable name

Assignment operator

Statement terminator

The area between { and } is a block

Return value

Function

Parameters

Expression

Functions

A function is a self-contained module of code that can be re-used.

You've been using the functions included with Processing such as `size()`, `line()`, and `stroke()` to write your programs, but it's also possible to write your own functions that make a program modular.

Functions make code more concise by extracting the common elements and making them into code blocks that can be run many times within the program. This makes the code easier to read and update and reduces the chance of errors.

Parameters

Functions can take no parameters, or they can take multiple parameters that modify their actions. Parameters are defined for functions inside round brackets after their name.

The `size()` function takes two parameters that define the display window and this is declared like this:

```
size(int width, int height)
```

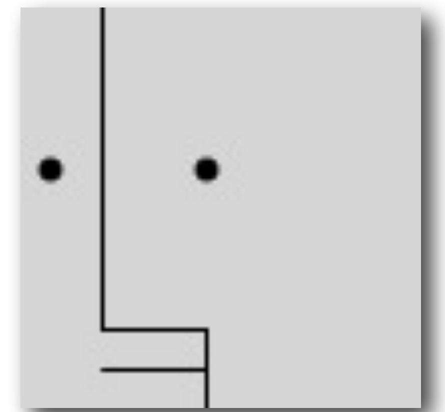
Functions can be defined with different numbers of parameters, by defining multiple functions with the same name.

A single parameter to the `fill()` function defines a grey value and three parameters defines an RGB colour.

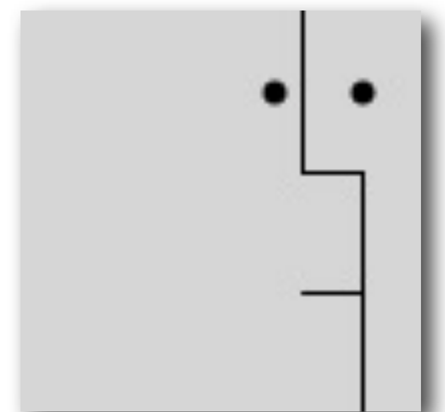
```
fill(float gray)
fill(float value1, float value2, float value3)
```

Parameterised Drawing

```
void setup() {
  size(100, 100);
  smooth();
  fill(0);
  face(20, 80, 26);
}

void face(int x, int y, int gap) {
  line(x, 0, x, y);               // Nose Bridge
  line(x, y, x+gap, y);           // Nose
  line(x+gap, y, x+gap, height);
  int mouthY = (height+y)/2;
  line(x, mouthY, x+gap, mouthY); // Mouth
  ellipse(x-gap/2, y/2, 5, 5);    // Left eye
  ellipse(x+gap, y/2, 5, 5);      // Right eye
}
```



face(20, 80, 26)



face(70, 40, 15)

Return Types

Functions may provide a value when they complete whatever task it is that they do, we call this the return value. Examples of functions that return values include the `random()` and the `sin()` functions that return values calculated based on parameters passed to them.

When a function is defined it will define the type of the values it returns. For example, one variant of the `random()` function is defined as:

```
float random(float low, float high)
```

Meaning that it takes takes two floating point values and returns a floating point value when it is done.

If a function does not return a value the function is declared as having a return type of `void`. For example, the single parameter version of the `background()` function doesn’t return anything, so it is declared as:

```
void background(float gray)
```

Returning Values

The keyword `return` is used to exit a function and return to the point in the program from which it was called.

When a function outputs a value, `return` is used to specify what value should be returned. The `return` statement is typically the last line of a function because functions exit immediately after a `return`.

If a function returns a value, the function almost always appears to the right of an assignment operator or as a part of a larger expression.

A function that does not return a value is often used as a complete statement.

If a function's return value is not used immediately, or assigned to a variable, the value will be lost.

Using Functions that Return Values

Return values can be stored in variables for later use

```
float d = random(0, 100);
ellipse(50, 50, d, d);
```

Variables used to store return values need to be of the correct type...

```
int d = random(0, 100);        // ERROR! random() returns floats
ellipse(50, 50, d, d);
```

...or we need to use other functions to convert return values to the type of the variable

```
int d = int(random(0, 100));  // int() converts the float value
ellipse(50, 50, d, d);
```

Writing Your Own Functions that Return Values

The following sketch defines a function called `average()` that calculates the average of two floating point numbers given to it and returns another floating point number with the value calculated.

```
void setup() {
  float f = average(12.0, 6.0);
  println(f);
}

float average(float num1, float num2) {
  float ave = (num1 + num2) / 2.0;
  return ave;
}
```

Writing Your Own Functions that Return Values

The following sketch defines a pair of functions that take a single floating point number representing the temperature in one unit of measurement and convert it to another:

```
void setup() {
  float c = celsius(451.0);
  println(c);
}

float celsius(float f) {
  float c = (f - 32.0) * (5.0/9.0);
  return c;
}

float fahrenheit(float c) {
  float f = c * (9.0/5.0) + 32.0;
  return f;
}
```

Creating a Function

Sometimes we might want to create a function so that we don't have to type as much in our main code.

For example, suppose that we want to create a sketch that draws a lot of circles, we might want to create a function that reduces the amount of typing we need to do like so:

```
void circle(float x, float y, float radius) {
  ellipse(x, y, radius*2, radius*2);
}
```

This would allow use to replace all calls to `ellipse()` where we would need to specify the same value for the 3rd and 4th parameters to draw circles with called to `circle()`, specifying just the value for `radius`.

Notice that this also reduces the possibility of making errors by specifying different parameters to `ellipse()` when they should always be the same to draw a circle.

The Setup and Draw Functions

Processing recognises the name of some functions as being special and will call them automatically. In particular, Processing will call a function called `setup()` once when a sketch is started and another functions called `draw()` repeatedly, whenever it needs to update the display.

A Processing sketch can have only one `setup()` function and one `draw()` function. When a sketch is run the code is executed in the following order:

1. Code outside `setup()` and `draw()` is executed, typically this code declares variables and defines constants.
2. Code inside `setup()` is run once.
3. Code inside `draw()` is run repeatedly until the program is stopped.

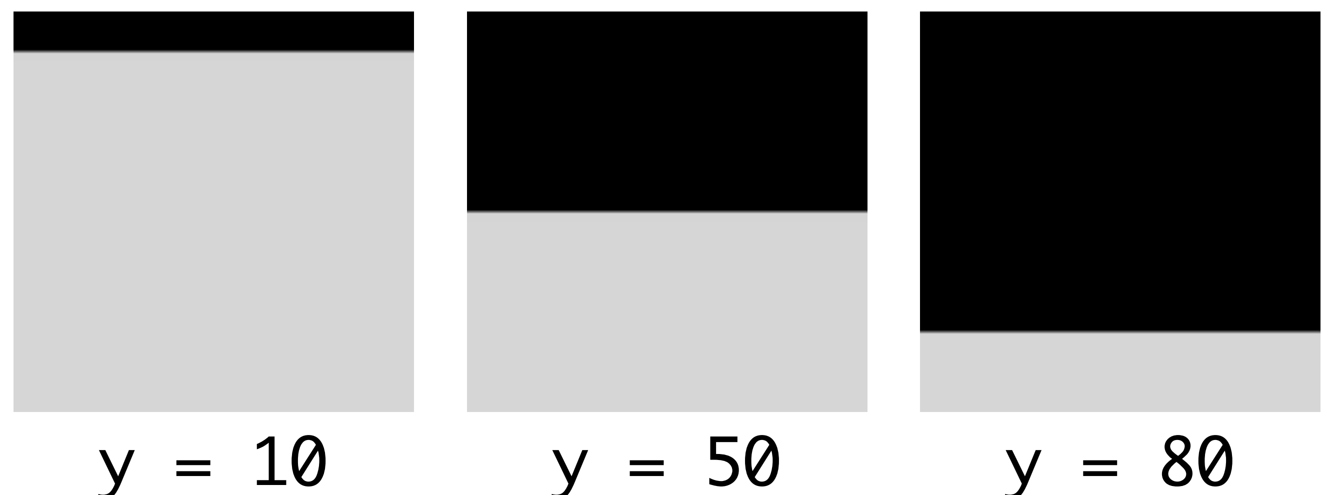
Continuous Drawing

The Draw Function

Programs that run continuously must include a `draw()` function.

The code inside a `draw()` block runs until the stop button is pressed or the window is closed. We can use this to draw continuously.

```
float y = 0.0;

void setup() {
  size(100, 100);
}

void draw() {
  line(0, y, 100, y);
  y = y + 0.5;
}
```



Frames

Processing calls each call to the `draw()` function a “frame”. The `frameCount` variable contains the total number of frames displayed.

```
void draw() {
  println(frameCount);
}
```

Output

```
1
2
3
4
5
6
```

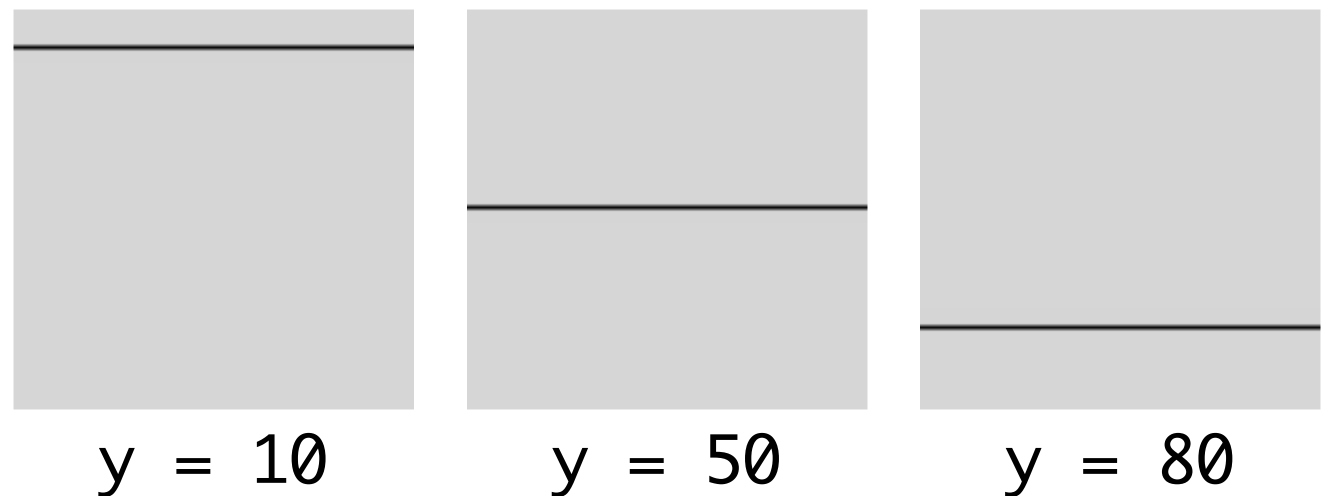
The `frameRate()` function changes the maximum number of frames per second.

The function controls only the maximum frame rate—it can not speed up a program that runs slowly because of equipment limitations.

Clearing the Background

By clearing the background at the start of every frame we can animate the line being drawn each time:

```
float y = 0.0;

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  line(0, y, 100, y);
  y = y + 0.5;
}
```



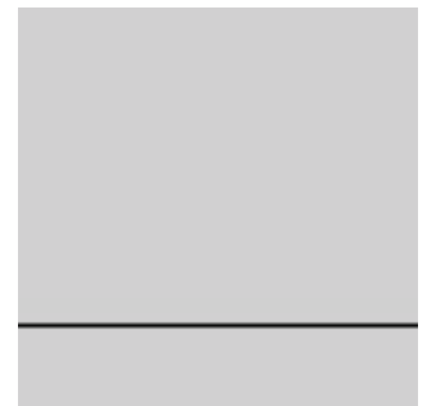
Animating the Background

By using the y variable to control the background colour as well as the line position we can animate that too.

```
float y = 0.0;

void setup() {
  size(100, 100);
}

void draw() {
  background(y * 2.5);
  line(0, y, 100, y);
  y = y + 0.5;
}
```

y = 10

y = 50



y = 80

Using Conditionals to Loop Animations

By checking to see that variables remain within the displayable area we can give the appearance of an animation looping:

```
float y = 0.0;

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  line(0, y, 100, y);
  y = y + 0.5;
  if (y > height) {
    y = 0.0;
  }
}
```

y = 10

y = 50

y = 80

Unrolling the Running of a Sketch

```
float y = 0.0
-------------------------------- Enter setup()
size(100, 100)
smooth()
fill(0)
-------------------------------- Enter draw()
background(204)
ellipse(50, 0.0, 70, 70)
y = 0.5
-------------------------------- Enter draw() for the 2rd time
background(204)
ellipse(50, 0.5, 70, 70)
y = 1.0
-------------------------------- Enter draw() for the 3rd time
background(204)
Etc.
```

Pausing an Animation

The `noLoop()` function stops your sketch from calling `draw()`.

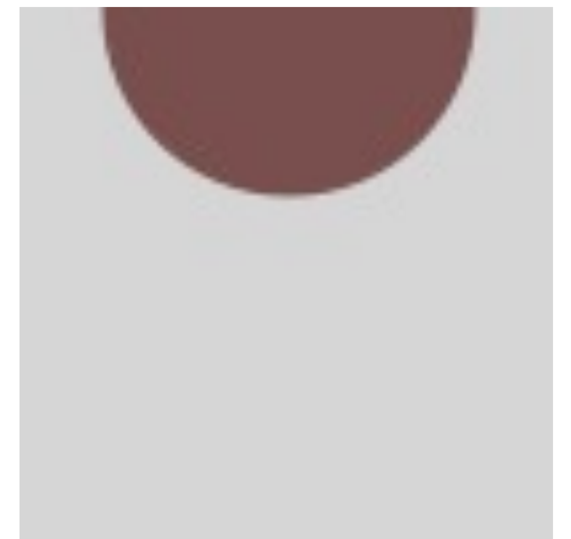
- Using `noLoop()` you can inspect your animation at specific points, e.g., the first frame.

It is possible to start a frozen animation using the `loop()` function.

- NOTE: The `loop()` function needs to be called from a function that responds to something outside of the animation. We'll get to that later...

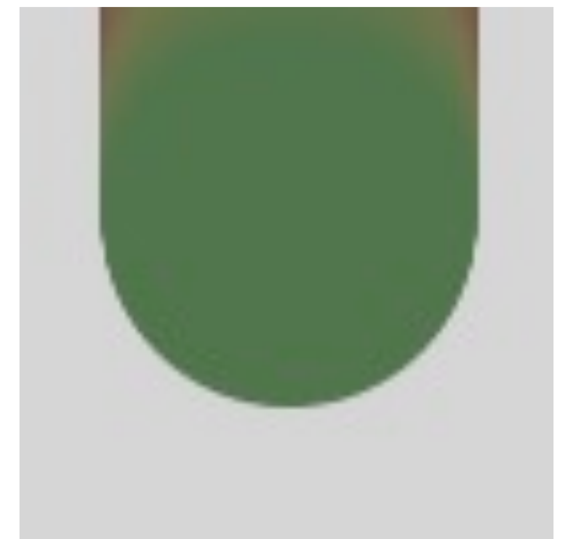
Inspecting the Start of an Animation

```
float y = 0.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  colorMode(HSB);
  noLoop();
}

void draw() {
  fill(frameCount % 255, 100, 100);
  ellipse(50, y, 70, 70);
  y += 0.5;
}
```



Using Conditionals to Inspect an Animation

```
float y = 0.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  colorMode(HSB);
}

void draw() {
  fill(frameCount % 255, 100, 100);
  ellipse(50, y, 70, 70);
  y += 0.5;
  if (y == 40) { noLoop(); }
}
```



Variable Scope

Every variable has a “scope”, i.e., a part of the program where it can be seen and used. When we start writing our own functions it is important to be aware of the scope of the variables we’re using:

```
int d = 51;         // Global variable d can be used everywhere

void setup() {
  size(100, 100);
  int val = d * 2; // Local variable val can only be used in setup()
  fill(val);
}

void draw() {
  int y = 60;      // Local variable y can only be used in draw()
  line(0, y, d, y);
}
```

Variable Scope with Conditions and Loops

```
void draw() {
  int d = 80;     // This variable can be used everywhere in draw()
  if (d > 50) {
    int x = 10;  // This variable can be used only in this if block
    line(x, 40, x+d, 40);
  }
  line(0, 50, d, 50);
  line(x, 60, x+d, 60); // ERROR! x can't be read outside block
}

void draw() {
  for (int y = 20; y < 80; y += 6) { // The variable y can be used
    line(20, y, 50, y);              // only within the for block
  }
  line(y, 0, y, 100); // ERROR! y can't be accessed outside for
}
```

Passing Arrays to Functions

New functions can be written to perform operations on arrays, but arrays behave differently than data types such as int and char.

When an array is passed to a function, the address (location in memory) of the array is transferred instead of the actual data. No new array is created, and changes made within the function affect the array used as the parameter.

```
float[] data = {19.0, 40.0, 75.0, 76.0, 90.0};

void setup() {
  halve(data);
  println(data); // Prints array containing "9.5 20.0 37.5 38.0 45.0"
}

void halve(float[] d) {
  for (int i = 0; i < d.length; i++) { // For each array element,
    d[i] = d[i] / 2.0;                 // divide the value by 2
  }
}
```

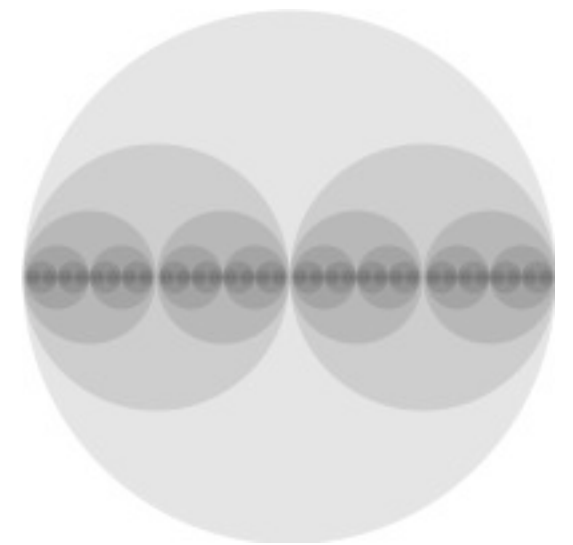
```
float[] data = {19.0, 40.0, 75.0, 76.0, 90.0};
float[] halfData;

void setup() {
  halfData = halve(data);                // Run the halve() function
  println(data[0] + ", " + halfData[0]); // Prints "19.0, 9.5"
  println(data[1] + ", " + halfData[1]); // Prints "40.0, 20.0"
  println(data[2] + ", " + halfData[2]); // Prints "75.0, 37.5"
  println(data[3] + ", " + halfData[3]); // Prints "76.0, 38.0"
  println(data[4] + ", " + halfData[4]); // Prints "90.0, 45.0"
}

float[] halve(float[] d) {
  float[] numbers = new float[d.length];     // Create a new array
  arraycopy(d, numbers);                     // Copy the contents
  for (int i = 0; i < numbers.length; i++) { // For each element,
    numbers[i] = numbers[i] / 2;             // divide the value by 2
  }
  return numbers;                            // Return the new array
}
```

Recursion

- Recursion is a common programming technique, where a function calls itself
  - Recursion is commonly used to process tree-like structures, e.g., HTML files
- Recursion can be used to draw complex fractal-like structures
  - Recursion makes drawing structures with self-similarity simple

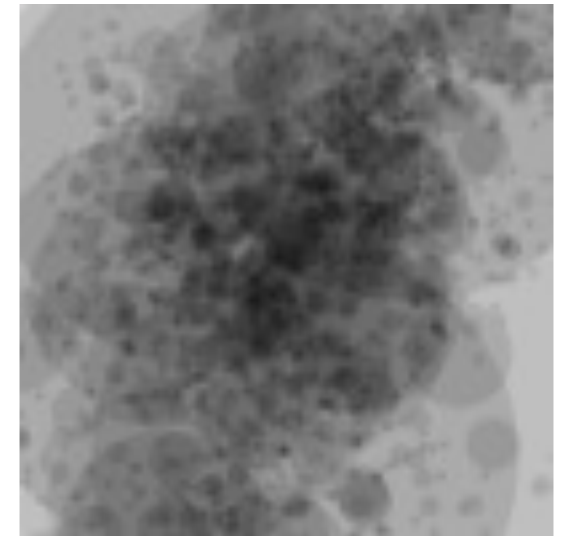
Drawing with Recursion

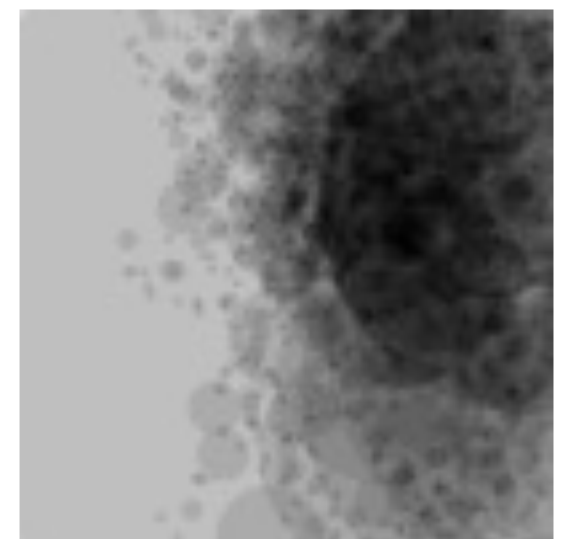
```
void setup() {
  size(200, 200);
  background(255);
  fill(0, 32);
  noStroke();
  smooth();
  circle(width/2, height/2, width/2, 10);
}

void circle(int x, int y, int r, int num) {
  ellipse(x, y, r*2, r*2);
  if (num > 0) {
    circle(x - r/2, y, r/2, num-1);
    circle(x + r/2, y, r/2, num-1);
  }
}
```



Drawing with Recursion

```
void setup() {
  size(200, 200);
  fill(0, 32);
  noStroke();
  circle(width/2, height/2, width/2, 10);
}

void circle(float x, float y, int r, int num) {
  ellipse(x, y, r*2, r*2);
  if (num > 0) {
    int branches = int(random(2, 6));
    for (int i = 0; i < branches; i++) {
      float a = random(0, TWO_PI);
      float newx = x + cos(a) * 6.0 * num;
      float newy = y + sin(a) * 6.0 * num;
      circle(newx, newy, r/2, num-1);
    }
  }
}
```





Time

Seconds, Minutes, Hours

Processing programs can read the value of the computer's clock.

- The current second is read with the `second()` function, returning 0–59.
- The current minute is read with the `minute()` function, returning 0–59.
- The current hour is read with the `hour()` function, returning 0–23.

```
int s = second(); // Returns values from 0 to 59
int m = minute(); // Returns values from 0 to 59
int h = hour();   // Returns values from 0 to 23
println("The time is " + h + ":" + m + ":" + s);
```

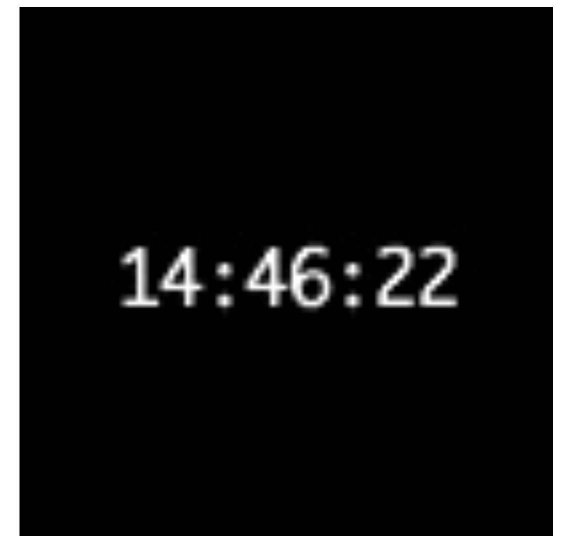
Telling the Time

```
int lastSecond = second();

void draw() {
  if (second() != lastSecond) {
    println(hour() + ":" + minute() + ":" + second());
    lastSecond = second();
  }
}
```

```
14:32:15
14:32:16
14:32:17
```

Drawing the Time

```
void setup() {
  textFont(loadFont("Monaco-14.vlw"));
  textAlign(CENTER, CENTER);
}

void draw() {
  background(0);
  int s = second();
  int m = minute();
  int h = hour();
  // The nf() function spaces the numbers nicely
  String t = nf(h,2) + ":" + nf(m,2) + ":" + nf(s,2);
  text(t, width/2, height/2);
}
```



Drawing the Time

```
void draw() {
  translate(width/2, height/2);
  background(0);
  noStroke();
  fill(80);
  // Angles for sin() and cos() start a 3 o'clock,
  // subtract HALF_PI to make them start at the top
  ellipse(0, 0, 80, 80);
  stroke(255);
  float s = map(second(), 0, 60, 0, TWO_PI) - HALF_PI;
  float m = map(minute(), 0, 60, 0, TWO_PI) - HALF_PI;
  float h = map(hour() % 12, 0, 12, 0, TWO_PI) - HALF_PI;
  line(0, 0, cos(s) * 38, sin(s) * 38);
  line(0, 0, cos(m) * 30, sin(m) * 30);
  line(0, 0, cos(h) * 25, sin(h) * 25);
}
```

14:03:26

Milliseconds

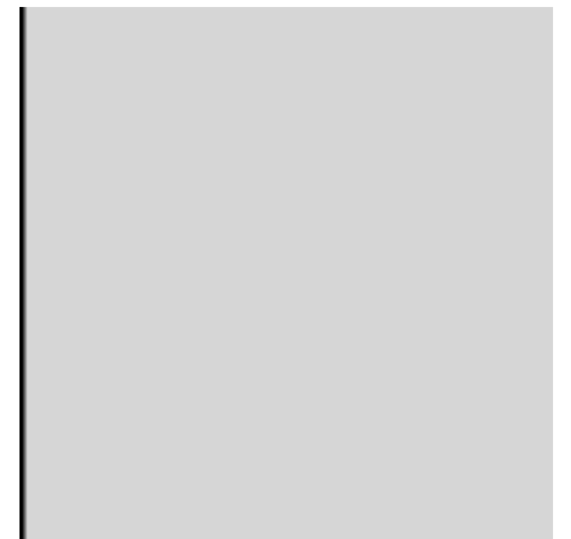
Each Processing program counts the time passed since the program started. This time is stored in milliseconds (thousandths of a second) and is obtained with the `millis()` function.

```
2000 milliseconds is 2 seconds
200 milliseconds is 0.2 seconds
```

Use the `millis()` function to trigger events (e.g. animations) and calculate the passage of time.

Controlling Animations

```
// Uses millis() to start a line in motion
// three seconds after the program starts
int x = 0;

void draw() {
  if (millis() > 3000) {
    x++;
  }
  line(x, 0, x, 100);
}
```



Timing Functions

```
void drawCircle(float x, float y, int r, int num) {
  int startMillis = millis();
  ellipse(x, y, r*2, r*2);
  if (num > 0) {
    int branches = int(random(2, 6));
    for (int i = 0; i < branches; i++) {
      float a = random(0, TWO_PI);
      float newx = x + cos(a) * 6.0 * num;
      float newy = y + sin(a) * 6.0 * num;
      drawCircle(newx, newy, r/2, num-1);
    }
  }
  println("drawCircle() took " + millis() - startMillis + "ms");
}
```