# 09 Making Methods

As we have seen from the previous sections, we can write simple instructions for the computer, and the computer will follow these instructions line-by-line. A computer will happily continue following these instructions for 10,000 lines or more.

However, as humans, we have trouble with long code. It become more complex, harder to understand, and harder to keep organized.

To make things simpler for us, it is good to group code together into logical chunks. These chunks can be easier to organize, and can make a program easier to understand. One way that we do this is by writing our own methods.

Creating our own functions allows us to organize our code into smaller chunks and treat complicated tasks as a single step.

We can also use predefined functions that Processing calls automatically do do things like perform animations and get user input.

## Example

Here is a simple program for creating a ball, having it move across the screen, and bounce off the walls.

```
int x;
int y;
int speed;

void setup() {
  size(800, 600);
  x = 0;
  y = height/2;
  speed = 5;
}

void draw() {
  background(0);
  fill(255, 0, 0);
  circle(x, y, 30);
  x = x + speed;

  if (x > width || x < 0) {
    speed = -speed;
  }
}
```

## Wouldn’t it be Nice If…

Wouldn’t it be nice if our code could look like this:

```
void draw() {
  background(0);
  drawBall();
  moveBall();
  bounceBall();
}
```

This is much easier to read and understand for us humans.

We’ll we can! We just need to make our own methods with these names. Here is what this program would look like with our code organized into methods.

```
int y;
int speed;

void setup() {
  x = 0;
  y = height/2;
  speed = 5;
  size(800, 600);
}

void draw() {
  background(0);
  drawBall();
  moveBall();
  bounceBall();
}

void drawBall() {
  fill(255, 0, 0);
  circle(x, y, 30);
}

void moveBall() {
   x = x + speed;
}

void bounceBall() {
    if (x > width || x < 0) {
    speed = -speed;
  }
}
```

## How To Make A Method

To make a method we need 4 things:

1. The "return type"—the expected **output**—of the method.
2. The name of the method.
3. Parenthesis, with a list of any parameters—the expected **input**—for the method
4. Curly braces containing the instructions—the **process**—of the method

So basically a method is saying, "I'll give you this, if you give me that, and this is how I'll do it."

## Method Output: Understanding Return Types

When the computer is finished with a method, sometimes it gives you a value back. Other times, it doesn't give you anything back.

For example, if you ask me to beat an egg to use in a cake, I will give you a bowl of beaten egg back. If you ask me to eat the cake, I'm not giving you anything back, because there's nothing left.

Similarly, methods that draw a circle, or print something to the screen give nothing back, so their **return type** is **void**, which means empty or nothing.

A method that adds numbers, or calculates an average will give you back a number (**int** or **float**). A method that requests a user to input their name, will return a **String**. A method that checks if a game is over will return a **boolean** (true or false).

## Method Examples

This method draws a red circle. It returns nothing (**void**).

```
void drawRedCircle(float circleX, float circleY, float circleDiameter){
  fill(255, 0, 0);
  circle(circleX, circleY, circleDiameter);
}
```

This method gives you the average of two numbers, and returns a number (**float**).

```
float average(float num1, float num2) {
  float answer = (num1 + num2) / 2;
  return answer;
}

float myAverage = average(15, 35);
```

This method tells you if a game is over.

```
boolean gameOver() {
 if (points <= 0) {
     return true;
 } else {
  return false;
 }
}
```

## Method Input: Understanding Parameters

(Example from https://happycoding.io/tutorials/processing/creating-functions )
Just like we can get values back from our methods, often we need to give our methods information. We already saw examples of this when drawing and coloring shapes. For example, we gave values to the `circle()` method to tell it where and how big to draw our circle.

When we create our own methods, we get to decide what information our method needs to do its job. Let's say we wanted to draw a target, we could make a method like this:

```
void drawTarget(float targetX, float targetY, float targetSize) {

  fill(255, 0, 0);
  circle(targetX, targetY, targetSize);

  fill(255, 255, 255);
  circle(targetX, targetY, targetSize*.75);

  fill(255, 0, 0);
  circle(targetX, targetY, targetSize/2);
}
```

Now, we can use this method over and over, just like the methods that are built into Processing.

```
void setup() {
  size(400, 400);
}

void draw() {
  drawTarget(100, 100, 200);
  drawTarget(300, 100, 100);
  drawTarget(100, 300, 150);
  drawTarget(300, 300, 175);
}

void drawTarget(float targetX, float targetY, float targetSize) {
  fill(255, 0, 0);
  circle(targetX, targetY, targetSize);
  fill(255, 255, 255);
  circle(targetX, targetY, targetSize*.75);
  fill(255, 0, 0);
  circle(targetX, targetY, targetSize/2);
}
```

We could also draw a target that follows the mouse:

```
void draw() {
  background(200);
  drawTarget(mouseX, mouseY, 100);
}
```

Or we could fill the screen up with random targets:

```
void draw() {
  drawTarget(random(0, width), random(0, height), random(25, 100));
}
```

## Things To Try

- Create a `drawHouse()` function that draws a house. Take in parameters for the house location, size, color, etc.
- Create a `drawBlock()` function that draws 4 houses. Take in parameters for the block location, size, color, etc. Don’t write code that draws 4 houses! Instead, call the `drawHouse()` function 4 different times with different parameters.
- Create a `drawNeighborhood()` function that draws 9 blocks. Take in parameters for the neighborhood location, size, color, etc. Call the `drawBlock()` function to draw the blocks.
- Create a `drawCity()` function that fills the window with neighborhoods.