The Imaginary Institute
www.imaginary-institute.com
2Dcourse@imaginary-institute.com

2D ANIMATION & INTERACTION COURSE
WEEK 4 QUICK REFERENCE

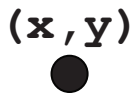
SHAPES

Each shape is created by calling its own particular function. As always, the difference between upper and lower case is important.

Points

```
point(x1, y1);
```

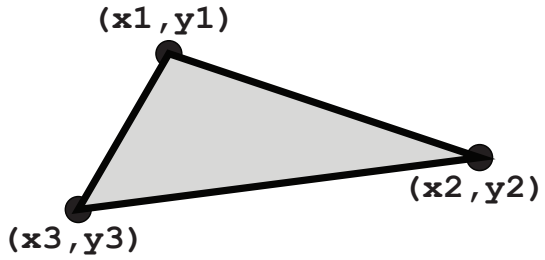
Draw a single point at `(x,y)`. The color of the point is the current color used for strokes.



Triangles

```
triangle(x1, y1, x2, y2, x3, y3);
```

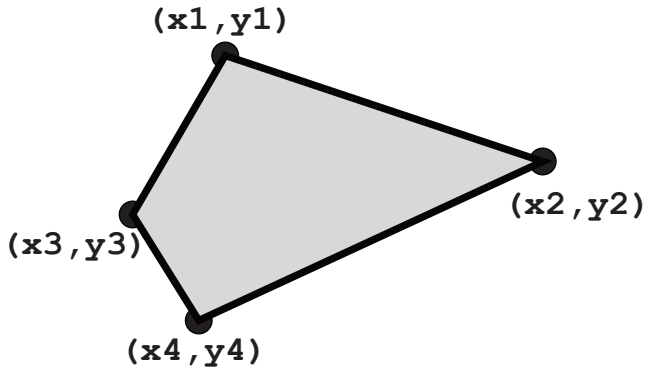
Draw a triangle joining the three points at `(x1,y1)`, `(x2,y2)`, and `(x3,y3)`.



Quads

`quad(x1, y1, x2, y2, x3, y3, x4, y4);`

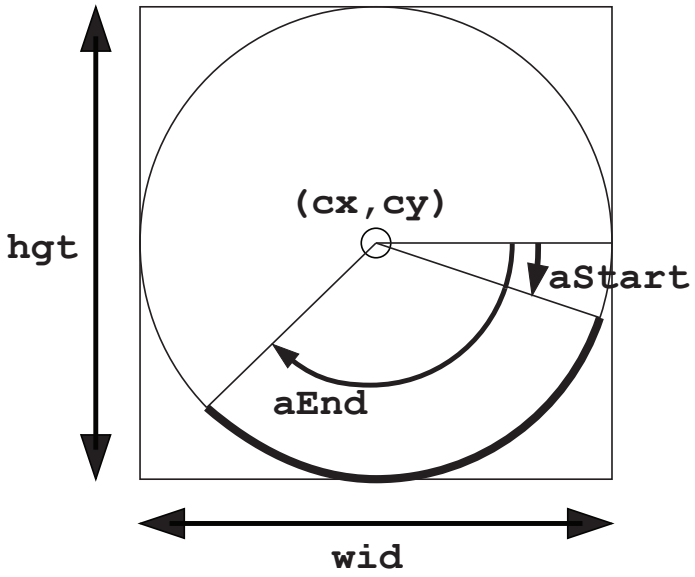
Draw a 4-sided joining the three points at `(x1,y1)`, `(x2,y2)`, `(x3,y3)`, and `(x4,y4)`.



Arcs

`arc(cx, cy, wid, hgt, aStart, aEnd);`

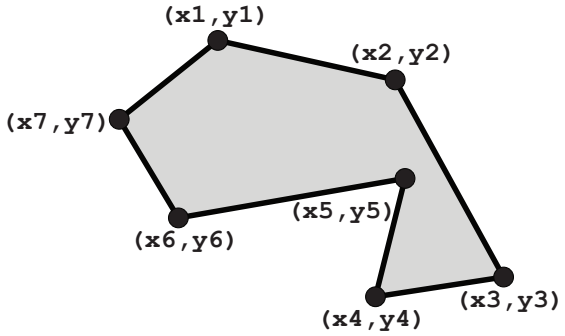
Draw an arc, or a section of a circle (or ellipse). The first four values define the ellipse. The last two give the starting and ending angles in radians, measured clockwise from 3 o'clock.



Polygons

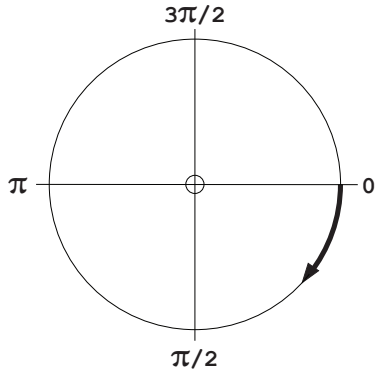
```
beginShape();
  vertex(x1, y1);
  vertex(x2, y2);
  ...
endShape();
```

Draw a multi-point polygon joining all the points given in the calls to `vertex()`, in order. To draw a line automatically from the last point to the first, give the keyword `CLOSE` as an argument to `endShape()`. I suggest getting in the habit of indenting all the lines between `beginShape()` and `endShape()` by one additional tab stop.



ANGLES

Angles are described as a value from 0 to 2π, where π has a value of about 3.14159. This value is saved in the keyword `PI`, and its related keywords `HALF_PI` and `TWO_PI`. An angle of 0 radians is at 3 o'clock, and values increase clockwise. If you have an angle in degrees, you can turn it into radians by handing it to the function `radians()`. To convert an angle in radians into degrees, hand it to `degrees()`.



LOOPS

There are two different styles for writing a loop. They are entirely equal in terms of what they can do, but often one form more closely matches the sequence of operations you're imagining as you write the code.

While Loops

A basic loop uses the keyword `while`, followed by a test in a pair of parentheses. As long as the value in the parentheses is true, the statement that follows will be executed over and over. To execute multiple statements, wrap them in curly braces. This example will draw a line of increasingly larger circles across the screen.

```
int circleX = 0;
while (circleX < 100) {
  circleX += 5;
  radius += 1;
  ellipse(circleX, 200, 2*radius, 2*radius);
}
```

For Loops

The above example has three key pieces: an *initialization* step, a *test*, and an *update* step. You can roll these all together into one line by using the `for` loop. The following loop produces the very same results as the while loop above:

```
for (int circleX=0; circleX <100; circle +=5 ) {
  radius += 1;
  ellipse(circleX, 200, 2*radius, 2*radius)
}
```

The `for` loop allows you to declare a variable in the initialization step, as shown here.

Skipping and Stopping

Sometimes it's useful to jump right back to the testing step of a loop, or to get out altogether. To jump immediately back to the test, and skip any remaining statements in the curly braces that make up the body of the loop, simply call `continue`. This statement does not have parentheses after it. To exit the loop immediately and jump immediately to the first line after the closing curly brace, call `break`. This also does not have parentheses after it.

```
for (int x=0; x<100; x+=2) {
  if (x == 50) break;  // skip only when x is 50
  ellipse(x, 100, 100, 100);
}
```

VARIATIONS ON IF

There are two variants on the `if` statement that are often convenient. Anything you can do with these, you can do with a normal `if` statement as well.

Switch

Choose from a number of alternative *case statements*. At the end you may have a `default` clause which gets executed if none of the others match. The values in the case statements must be constants (that is, they cannot be variables whose values can change). Each clause except the last usually ends with a `break` statement.

```
switch (numberOfPlayers) {
  case 0:
    println("You must have at least one player!");
    break;
  case 1:
    println("Welcome to the solo game.");
    break;
  default:
    println("Two or more players?  Let's go!");
 }
```

Conditional

The conditional offers a super-concise test for those times when everything is very short and you want to squeeze an if test and both clauses on one line. The test appears before the mark, followed by the true clause, a colon, and the false clause.

```
// Get 50 points if distance to target is
// within 10 units, otherwise no points.
score = distance < 10 ? 50 : 0;
```

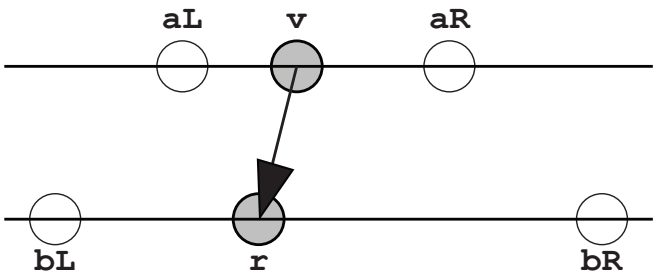
UTILTIES

Three built-in utilities for changing numbers are useful in many situations. You'll use these a lot when you work with the mouse.

map

Convert a number from a value within one range to a corresponding value within another range.

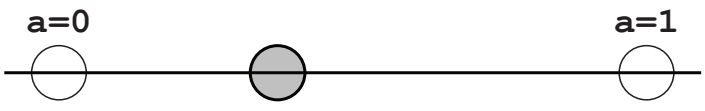
```
r = map(v, aL, aR, bL, bR);
```



lerp

Find a number that blends from one value to another. The float that controls the blend, here `a`, should be between 0 and 1 to get back values between the extremes. Note that unlike `map()`, the controlling value `a` appears at the *end* of the argument list.

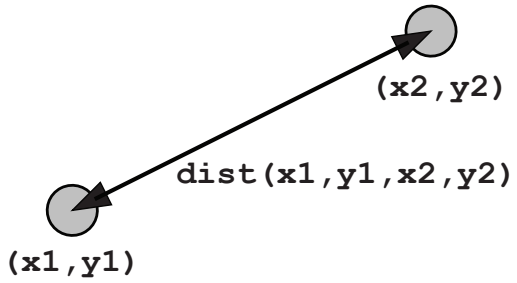
```
r = lerp(loValue, hiValue, a);
```



dist

Find the distance in pixels between points `(x1, y1)` and `(x2, y2)`.

```
d = dist(x1, y1, x2, y2);
```



THE MOUSE

The current location of the mouse in the graphics window is always given by the system variables `mouseX` and `mouseY`.

You can write a variety of routines that will be called automatically by the system. They should all be declared of type `void`, and none take arguments. Most of the time you'll keep these routines very short and very fast. Typically the goal is to change one or more global variables so that the next time `draw()` is called, the picture that's created is somehow responsive to the mouse.

Here are the most commonly useful mouse routines, along with the condition that causes them to be called:

`mousePressed()`	any mouse button was pressed
`mouseReleased()`	any mouse button was released
`mouseDragged()`	mouse moved while a button was down
`mouseMoved()`	mouse moves while no buttons were down

You can also test the value of a system-wide global variable to determine which mouse button is currently down:

`mouseButton`	has value `LEFT`, `RIGHT`, or `CENTER`